EECS 370

Midterm Review

What do you want me to prioritize today?

- https://forms.gle/sfJrbZsmEyFrf9e56

https://forms.gle/sfJrbZsmEyFrf9e56

Exam Information

- **Thursday, 10/12 at 6-8 PM EST - In Person**
- Coverage: up to and including lecture 11 (“Multi-Cycle Processor Design”)
- What to bring:
  - #2 pencil OR black/blue pen
  - One 8.5x11 double-sided cheat sheet
  - MCard
  - basic/scientific calculator or graphing calculator that meets SAT standards
- Look at @1421 on piazza for more logistics information

https://forms.gle/sfJrbZsmEyFrf9e56

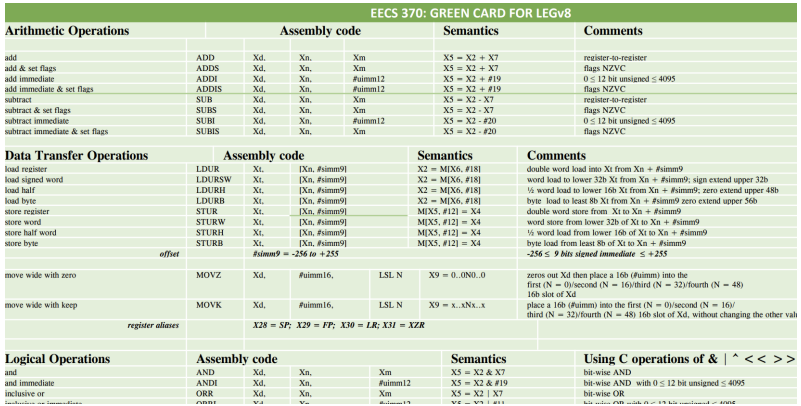
Exam Prep Tips

- Go over Lab slides
- Redo problems from the Homeworks/Labs
- Go through the Past Exams
- Come to Office Hours!
- **Please take care of your health.** (Sleep well, eat smart, take breaks, etc.)
- Today: ASK QUESTIONS. We will take all the time we need to make sure everyone is on the same page.
- Remember, this isn't a math class. **Conceptual understanding** is more important than how many formulas you've memorized.

Reference Material

https://forms.gle/sfJrbZsmEyFrf9e56

- https://eecs370.github.io/resources/armrefsheet.pdf
- https://docs.google.com/document/d/1b1hCLoDanjWy56IMqFMtbuHEv-mSFNGF7sLNkCOcRPI/edit#heading=h.vzaojd947c



EECS 370: GREEN CARD FOR LEGv8

Arithmetic Operations	Assembly code				Semantics	Comments
add	ADD	Xd,	Xn,	Xm	X5 = X2 + X7	register-to-reigister
add & set flags	ADDS	Xd,	Xn,	Xm	X5 = X2 + X7	flags NZVC
add immediate	ADDI	Xd,	Xn,	#uimm12	X5 = X2 + #19	0 ≤ 12 bit unsigned ≤ 4095
add immediate & set flags	ADDIS	Xd,	Xn,	#uimm12	X5 = X2 + #19	flags NZVC
subtract	SUB	Xd,	Xn,	Xm	X5 = X2 - X7	register-to-register
subtract & set flags	SUBS	Xd,	Xn,	Xm	X5 = X2 - X7	flags NZVC
subtract immediate	SUBI	Xd,	Xn,	#uimm12	X5 = X2 - #20	0 ≤ 12 bit unsigned ≤ 4095
subtract immediate & set flags	SUBIS	Xd,	Xn,	Xm	X5 = X2 - #20	flags NZVC

Data Transfer Operations	Assembly code			Semantics	Comments
load register	LDUR	Xt,	[Xn, #simm9]	X2 = M[X6, #18]	double word load into Xt from Xn + #simm9
load signed word	LDURSW	Xt,	[Xn, #simm9]	X2 = M[X6, #18]	word load to lower 32b Xt from Xn + #simm9; sign extend upper 32b
load half	LDURH	Xt,	[Xn, #simm9]	X2 = M[X6, #18]	½ word load to lower 16b Xt from Xn + #simm9; zero extend upper 48b
load byte	LDURB	Xt,	[Xn, #simm9]	X2 = M[X6, #18]	byte load to least 8b Xt from Xn + #simm9 zero extend upper 56b
store register	STUR	Xt,	[Xn, #simm9]	M[X5, #12] = X4	double word store from Xt to Xn + #simm9
store word	STURW	Xt,	[Xn, #simm9]	M[X5, #12] = X4	word store from lower 32b of Xt to Xn + #simm9
store half word	STURH	Xt,	[Xn, #simm9]	M[X5, #12] = X4	½ word load from lower 16b of Xt to Xn + #simm9
store byte	STURB	Xt,	[Xn, #simm9]	M[X5, #12] = X4	byte load from least 8b of Xt to Xn + #simm9
offset		***#simm9 = -256 to +255***			***-256 ≤ 9 bits signed immediate ≤ +255***
move wide with zero	MOVZ	Xd,	#uimm16,	LSL N	X9 = 0..0N0..0	zeros out Xd then place a 16b (#uimm) into the first (N = 0)/second (N = 16)/third (N = 32)/fourth (N = 48) 16b slot of Xd
move wide with keep	MOVK	Xd,	#uimm16,	LSL N	X9 = x..xNx..x	place a 16b (#uimm) into the first (N = 0)/second (N = 16)/ third (N = 32)/fourth (N = 48) 16b slot of Xd, without changing the other val
register aliases		***X28 = SP; X29 = FP; X30 = LR; X31 = XZR***				

Logical Operations	Assembly code				Semantics	Using C operations of & | ^ << >>
and	AND	Xd,	Xn,	Xm	X5 = X2 & X7	bit-wise AND
and immediate	ANDI	Xd,	Xn,	#uimm12	X5 = X2 & #19	bit-wise AND with 0 ≤ 12 bit unsigned ≤ 4095
inclusive or	ORR	Xd,	Xn,	Xm	X5 = X2 | X7	bit-wise OR

EECS 370 Exam Reference Packet

Assume these data-type sizes (unless specified otherwise)

Type	Storage size
char	1 byte
int	4 bytes
short	2 bytes
long	8 bytes

LEGv8 Application Binary Interface

- X30: **link register**
- X28: **stack pointer**
- X19-X27: **callee-saved**
- X0-15: **caller-saved**
- X0-X7: **arguments** (memory used if more space is needed)
- X0: **return value**

Topics Covered

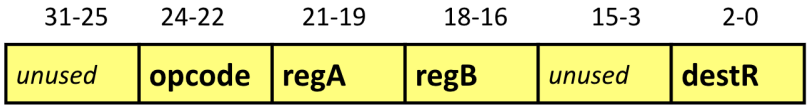
- **Understanding the LC2K ISA**
- ARM
- Endianness
- Linking/Linker
- Caller/Callee
- LC2K Processor Performance
  - Single-Cycle Datapath
  - Multi-Cycle Datapath
- Pipeline Datapath

https://forms.gle/sfJrbZsmEyFrf9e56

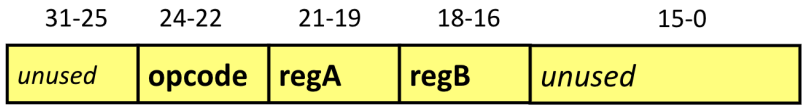
Understanding the LC2k ISA

- Understand what each LC2k Instruction does, see project 1 spec for a useful chart breaking down their functions
- Remember how LC2k instructions are broken down
- Be able to simulate an LC2K assembly program and answer questions about its operation

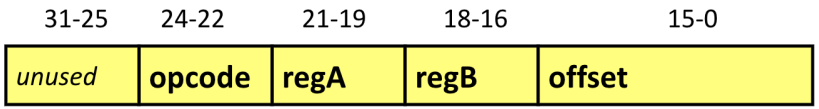
R type instructions (add ‘000’, nor ‘001’)



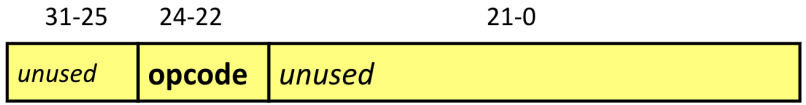
J-type instructions (jalr ‘101’)



I type instructions (lw ‘010’, sw ‘011’, beq ‘100’)



O type instructions (halt ‘110’, noop ‘111’)



Understanding the LC2k ISA

- LC2k is word addressable, 32 bit instructions, 8 registers
- Difference between:

```
        lw    0    4    ArrAdr          vs          lw    0    4    Arr
ArrAdr    .fill    Arr                          ArrAdr    .fill    Arr
Arr       .fill    0                       Arr       .fill    0
          .fill    3                                 .fill    3
          .fill    4                                 .fill    4
```

- Looping in LC2k with beq
- Function calls with jalr

Topics Covered

- Understanding the LC2K ISA
- **ARM**
- Endianness
- Linking/Linker
- Caller/Callee
- LC2K Processor Performance
  - Single-Cycle Datapath
  - Multi-Cycle Datapath

C to ARM: Loops and If Statements

Both are implemented with branches and comparisons

For loops, we want to repeat segments of code. We can accomplish this by branching up (to previous lines of code).

For if statements, we want to be able to skip segments of code depending on the condition. We can accomplish this by branching down (to further lines of code)

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
movz    x2,   #0
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
movz    x2,  #0
movz    x1,  #0
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
        movz    x2,  #0
        movz    x1,  #0
loop    cmpi    x1,  #10
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
      movz    x2,  #0
      movz    x1,  #0
loop  cmpi    x1,  #10
      b.ge    done
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
      movz    x2,  #0
      movz    x1,  #0
loop  cmpi    x1,  #10
      b.ge    done
      cmpi    x1,  #3
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
      movz    x2,  #0
      movz    x1,  #0
loop  cmpi    x1,  #10
      b.ge    done
      cmpi    x1,  #3
      b.ge    skip1
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
      movz    x2,  #0
      movz    x1,  #0
loop  cmpi    x1,  #10
      b.ge    done
      cmpi    x1,  #3
      b.ge    skip1
      add     x2,  x1,  x2
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
    movz    x2,  #0
    movz    x1,  #0
loop    cmpi    x1,  #10
    b.ge    done
    cmpi    x1,  #3
    b.ge    skip1
    add     x2,  x1,  x2  skip1  andi    x3,
x1,  #3
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
    movz    x2,  #0
    movz    x1,  #0
loop    cmpi    x1,  #10
    b.ge    done
    cmpi    x1,  #3
    b.ge    skip1
    add     x2,  x1,  x2  skip1  andi    x3,
x1,  #3
    cbnz    x3,  inc
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
    movz    x2, #0
    movz    x1, #0
loop    cmpi    x1, #10
    b.ge    done
    cmpi    x1, #3
    b.ge    skip1
    add     x2, x1, x2  skip1   andi    x3,
x1, #3
    cbnz    x3, inc
    mul     x2, x1, x2
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
      movz    x2,  #0
      movz    x1,  #0
loop  cmpi    x1,  #10
      b.ge    done
      cmpi    x1,  #3
      b.ge    skip1
      add     x2,  x1,  x2  skip1  andi    x3,
  x1, #3
      cbnz    x3,  inc
      mul     x2,  x1,  x2
inc   addi    x1,  x1,  #1
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
    movz    x2,  #0
    movz    x1,  #0
loop    cmpi    x1,  #10
    b.ge    done
    cmpi    x1,  #3
    b.ge    skip1
    add     x2,  x1,  x2  skip1  andi    x3,
  x1,  #3
    cbnz    x3,  inc
    mul     x2,  x1,  x2
inc     addi  x1,  x1,  #1
    b     loop
```

Translating from C to ARM - Control Flow

Convert the following code to ARM:

```
int x2 = 0;
for (int x1 = 0; x1 < 10; ++x1) {
  if (x1 < 3) {
    x2 += x1;
  }
  if (x1 % 4 == 0) {
    x2 *= x1;
  }
}
```

```
      movz    x2,  #0
      movz    x1,  #0
loop  cmpi    x1,  #10
      b.ge    done
      cmpi    x1,  #3
      b.ge    skip1
      add     x2,  x1,  x2  skip1  andi    x3,
  x1, #3
      cbnz    x3,  inc
      mul     x2,  x1,  x2
inc   addi    x1,  x1,  #1
      b       loop
done
```

C to ARM: Structs

We access struct elements by going into memory (loads and stores)

Need starting address of struct element. Also need the offset within the struct that the specific element is at.

This requires us to first determine the memory alignment of the struct (padding will affect this)

Translating from C to ARM - Memory Access

Convert the following code to ARM:

```
//Assume it is 64 bit architecture

MyStruct* ptr; //ptr is stored in X10

ptr->a = X1;

ptr->d = &X2; //Assume the address for X2 is in X20

char X3 = ptr->b
```

```
struct {
  int a;

  char b;

  short c;

  int* d;
} MyStruct;
```

Translating from C to ARM - Memory Access

Convert the following code to ARM:

```
//Assume it is 64 bit architecture

MyStruct* ptr; //ptr is stored in X10

ptr->a = X1;

ptr->d = &X2; //Assume the address for X2 is in X20

char X3 = ptr->b
```

```
struct {
  int a; //0-3

  char b; //4-4

  short c; //6-7

  int* d; //8-15
} MyStruct;
```

Translating from C to ARM - Memory Access

Convert the following code to ARM:

```
//Assume it is 64 bit architecture

MyStruct* ptr; //ptr is stored in X10

ptr->a = X1;

ptr->d = &X2; //assume the address for X2 is in X20

char X3 = ptr->b
```

```
struct {
  int a;
  char b;
  short c;
  int* d;
} MyStruct;
```

```
sturw   X1,   [X10, #0]
stur    X20,  [X10, #8]
ldurb   X3,   [X10, #4]
```

C to ARM: Arrays

Like structs, we access array elements by going into memory (loads and stores)

Need starting address of the array. We also need the **memory offset** from the start of the array to the element we want.

Memory offset != Index. Each element takes up >= 1 Memory Address. Therefore, our Memory Offset = Index*Size of each element

Generally, we can compute this multiplication with a left shift

Translating from C to ARM - Arrays

Convert the following code to ARM:

```
//a -> X5; b -> X7
int64_t Hershal[100]; //Hershal -> X9

Hershal[b++] = Hershal[5]+a;
```

Translating from C to ARM - Arrays

Convert the following code to ARM:

```
//a -> X5; b -> X7
int64_t Hershal[100]; //Hershal -> X9

Hershal[b++] = Hershal[5]+a;
```

LDUR X1, [X9, #40] **<- We use 40 as the offset because 5*8=40**

Translating from C to ARM - Arrays

Convert the following code to ARM:

```
//a -> X5; b -> X7
int64_t Hershal[100]; //Hershal -> X9

Hershal[b++] = Hershal[5]+a;
```

```
LDUR   X1,   [X9,   #40]
ADD      X1,   X1,   X5    <- X1 now has Hershal[5]+a (notice how we can reuse temp registers)
```

Translating from C to ARM - Arrays

Convert the following code to ARM:

```
//a -> X5; b -> X7
int64_t Hershal[100]; //Hershal -> X9

Hershal[b++] = Hershal[5]+a;
```

```
LDUR   X1,   [X9,   #40]
ADD     X1,   X1,   X5
LSL     X2,   X7,   #3    <- Computes b*8 and stores it in X2 (b << 3 == b*8)
```

Translating from C to ARM - Arrays

Convert the following code to ARM:

```
//a -> X5; b -> X7
int64_t Hershal[100]; //Hershal -> X9

Hershal[b++] = Hershal[5]+a;
```

```
LDUR   X1,   [X9,   #40]
ADD     X1,   X1,   X5
LSL     X2,   X7,   #3
ADD     X2,   X9,   X2
```

<- This calculates the absolute memory address of Hershal[b]

Translating from C to ARM - Arrays

Convert the following code to ARM:

```
//a -> X5; b -> X7
int64_t Hershal[100]; //Hershal -> X9

Hershal[b++] = Hershal[5]+a;
```

```
LDUR   X1,   [X9,   #40]
ADD     X1,   X1,   X5
LSL     X2,   X7,   #3
ADD     X2,   X9,   X2
STUR   X1,   [X2,   #0]
```

<- Stores the result (X1) in Hershal[b]. Because we calculated the absolute memory address in the ADD (X2), we can set the immediate offset here to 0.

Translating from C to ARM - Arrays

Convert the following code to ARM:

```
//a -> X5; b -> X7
int64_t Hershal[100]; //Hershal -> X9

Hershal[b++] = Hershal[5]+a;
```

```
LDUR   X1,   [X9,   #40]
ADD     X1,   X1,   X5
LSL     X2,   X7,   #3
ADD     X2,   X9,   X2
STUR   X1,   [X2,   #0]
ADDI    X7,   X7,   #1
```

<- Don't forget the post-increment! (How would this problem change with a pre-increment?)

You're a master if you can solve this:

C Source Code

```
struct Contender {
   char  name[10];
   short score1;           // Scores can be negative
   int   scoreFinal;
};
int whoWon(struct Contender contenders[], int size){
   int total = 0, maxScore = 0x80000000, maxScoreIndex = 0, i = 0;
   for(i = 0; i < size; ++i){
        total = contenders[i].score1 + contenders[i].scoreFinal;
        if(total >= maxScore){
            maxScore = total;
            maxScoreIndex = i;
         }
    }

}
```

- contenders -> X0
- size -> X1
- total -> X2
- maxScore -> X3
- maxScoreIndex -> X4
- i -> X5
- Rest can be used as temp registers
- Don't worry about the function return

Topic Covered

- Understanding the LC2K ISA
- ARM
- **Endianness**
- Linking/Linker
- Caller/Callee
- LC2K Processor Performance
  - Single-Cycle Datapath
  - Multi-Cycle Datapath
- Pipeline Datapath

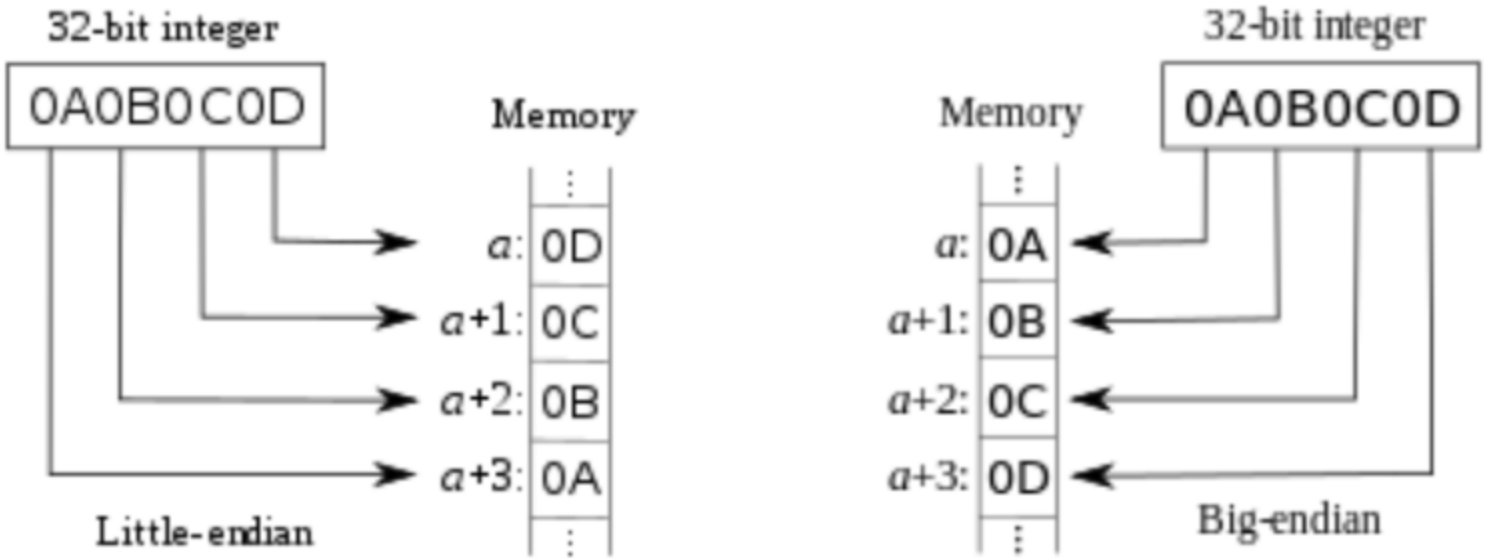
Endianness

Endianness is the ordering of bytes within a word.

- Little Endian
  - Increasing numeric significance with increasing memory addresses
- Big Endian
  - Opposite; most significant byte comes first
- Endianness will refer to ordering of bytes within a word IN MEMORY - think what is the MSB in memory

Endianness

Most useful diagram!!! Save it for reference



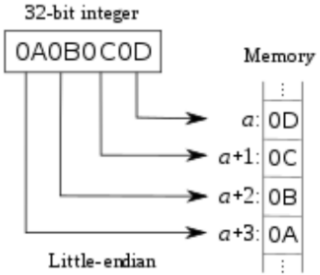
Endianness - Little Endian 32-bit architecture

Memory:

10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x44
16	0xA3

Registers:

W1	0x10	
W2	0x0	
W3	0x23	



Endianness - Little Endian 32-bit architecture

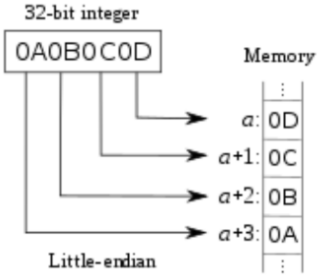
Memory:

10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x44
16	0xA3

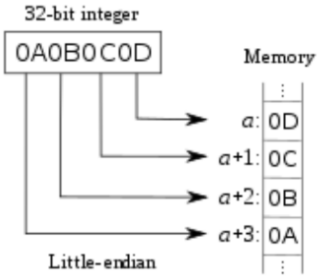
Registers:

W1	0x10
W2	0x0
W3	~~0x23~~

LDURSW W2 [W1, #2]



Endianness - Little Endian 32-bit architecture



Memory:

10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x44
16	0xA3

Registers:

W1	0x10
W2	0x44000123
W3	0x23

LDURSW W2 [W1, #2]

Endianness - Little Endian 32-bit architecture

Memory:

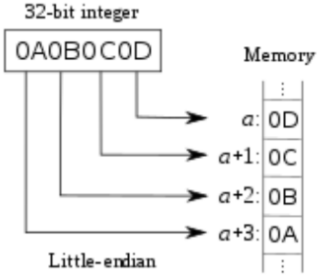
10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x44
16	0xA3

Registers:

W1	0x10	
W2	0x44000123	
W3	0x23	

LDURSW W2 [W1, #2]

LDURH W3 [W1, #2]



Endianness - Little Endian 32-bit architecture

Memory:

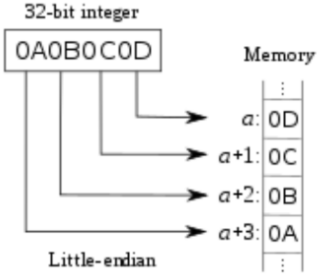
10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x44
16	0xA3

Registers:

W1	0x10
W2	0x44000123
W3	0x0123

LDURSW W2 [W1, #2]

LDURH W3 [W1, #2]



Endianness - Little Endian 32-bit architecture

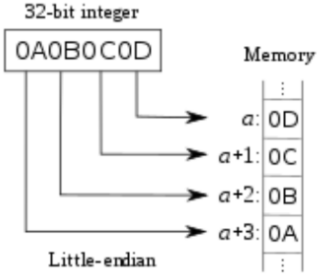
Memory:

10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x44
16	0xA3

Registers:

W1	0x10
W2	0x44000123
W3	0x0123

```
LDURSW   W2 [W1, #2]
LDURH    W3 [W1, #2]
STURH    W2 [W1, #5]
```



Endianness - Little Endian 32-bit architecture

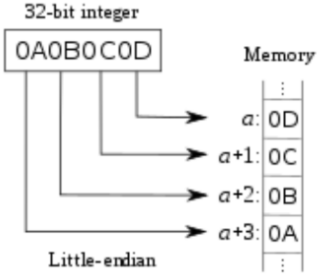
Memory:

10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x23
16	0x01

Registers:

W1	0x10
W2	0x44000123
W3	0x0123

```
LDURSW   W2 [W1, #2]
LDURH    W3 [W1, #2]
STURH    W2 [W1, #5]
```



Endianness - Little Endian 32-bit architecture

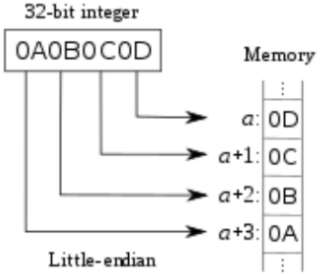
Memory:

10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x23
16	0x01

Registers:

W1	0x10
W2	0x44000123
W3	0x0123

```
LDURSW   W2 [W1, #2]
LDURH    W3 [W1, #2]
STURH    W2 [W1, #5]
```



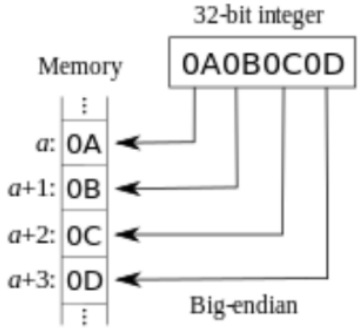
Endianness - Big Endian 32-bit architecture

Memory:

10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x44
16	0xA3

Registers:

W1	0x10
W2	0x0
W3	~~0x23~~



Endianness - Big Endian 32-bit architecture

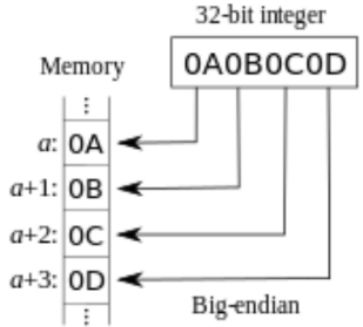
Memory:

10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x44
16	0xA3

Registers:

W1	0x10
W2	0x0
W3	0x23

LDURSW W2 [W1, #2]



Endianness - Big Endian 32-bit architecture

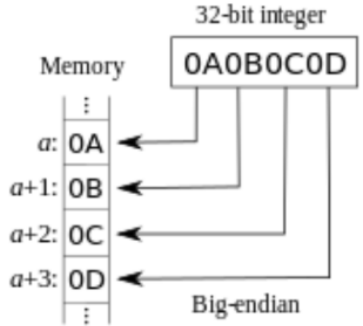
Memory:

10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x44
16	0xA3

Registers:

W1	0x10
W2	0x23010044
W3	~~0x23~~

LDURSW W2 [W1, #2]



Endianness - Big Endian 32-bit architecture

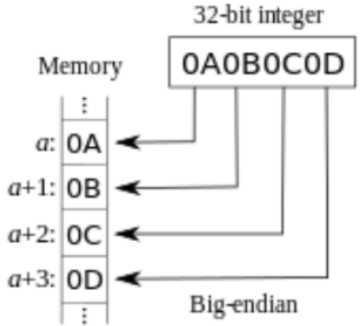
Memory:

10	0x0A	W1	0x10	
11	0xF2			
12	0x23	W2	0x23010044	
13	0x01			
14	0x00			
15	0x44	W3	0x23	
16	0xA3			

Registers:

LDURSW W2 [W1, #2]

LDURH W3 [W1, #2]



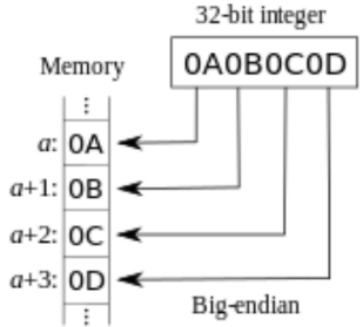
Endianness - Big Endian 32-bit architecture

Memory:

		Registers:		
10	0x0A	W1	0x10	
11	0xF2			
12	0x23	W2	0x23010044	
13	0x01			
14	0x00			
15	0x44	W3	~~0x2301~~	
16	0xA3			

LDURSW W2 [W1, #2]

LDURH W3 [W1, #2]



Endianness - Big Endian 32-bit architecture

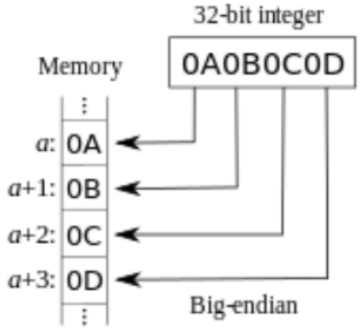
Memory:

10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x44
16	0xA3

Registers:

W1	0x10
W2	0x23010044
W3	~~0x2301~~

LDURSW W2 [W1, #2]
LDURH W3 [W1, #2]
STURH W2 [W1, #5]



Endianness - Big Endian 32-bit architecture

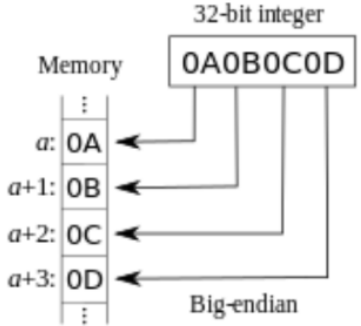
Memory:

10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x00
16	0x44

Registers:

W1	0x10
W2	0x23010044
W3	0x2301

LDURSW W2 [W1, #2]
LDURH W3 [W1, #2]
STURH W2 [W1, #5]



Endianness - Big Endian 32-bit architecture

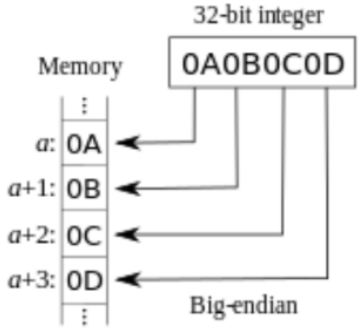
Memory:

10	0x0A
11	0xF2
12	0x23
13	0x01
14	0x00
15	0x00
16	0x44

Registers:

W1	0x10
W2	0x23010044
W3	0x2301

```
LDURSW   W2 [W1, #2]
LDURH    W3 [W1, #2]
STURH    W2 [W1, #5]
```

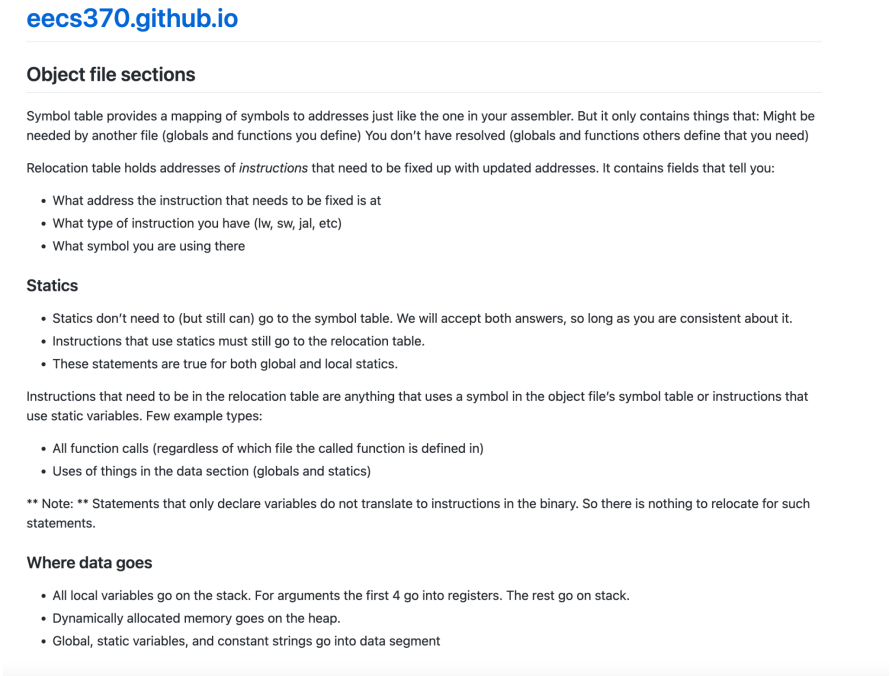


Topic Covered

- Understanding the LC2K ISA
- ARM
- Endianness
- **Linking/Linker**
- Caller/Callee
- LC2K Processor Performance
  - Single-Cycle Datapath
  - Multi-Cycle Datapath
- Pipeline Datapath
- CheatSheet Stuff

Course Resources

https://eecs370.github.io/resources/symbolTableGuide



eecs370.github.io

Object file sections

Symbol table provides a mapping of symbols to addresses just like the one in your assembler. But it only contains things that: Might be needed by another file (globals and functions you define) You don't have resolved (globals and functions others define that you need)

Relocation table holds addresses of *instructions* that need to be fixed up with updated addresses. It contains fields that tell you:

- What address the instruction that needs to be fixed is at
- What type of instruction you have (lw, sw, jal, etc)
- What symbol you are using there

Statics

- Statics don't need to (but still can) go to the symbol table. We will accept both answers, so long as you are consistent about it.
- Instructions that use statics must still go to the relocation table.
- These statements are true for both global and local statics.

Instructions that need to be in the relocation table are anything that uses a symbol in the object file's symbol table or instructions that use static variables. Few example types:

- All function calls (regardless of which file the called function is defined in)
- Uses of things in the data section (globals and statics)

** Note: ** Statements that only declare variables do not translate to instructions in the binary. So there is nothing to relocate for such statements.

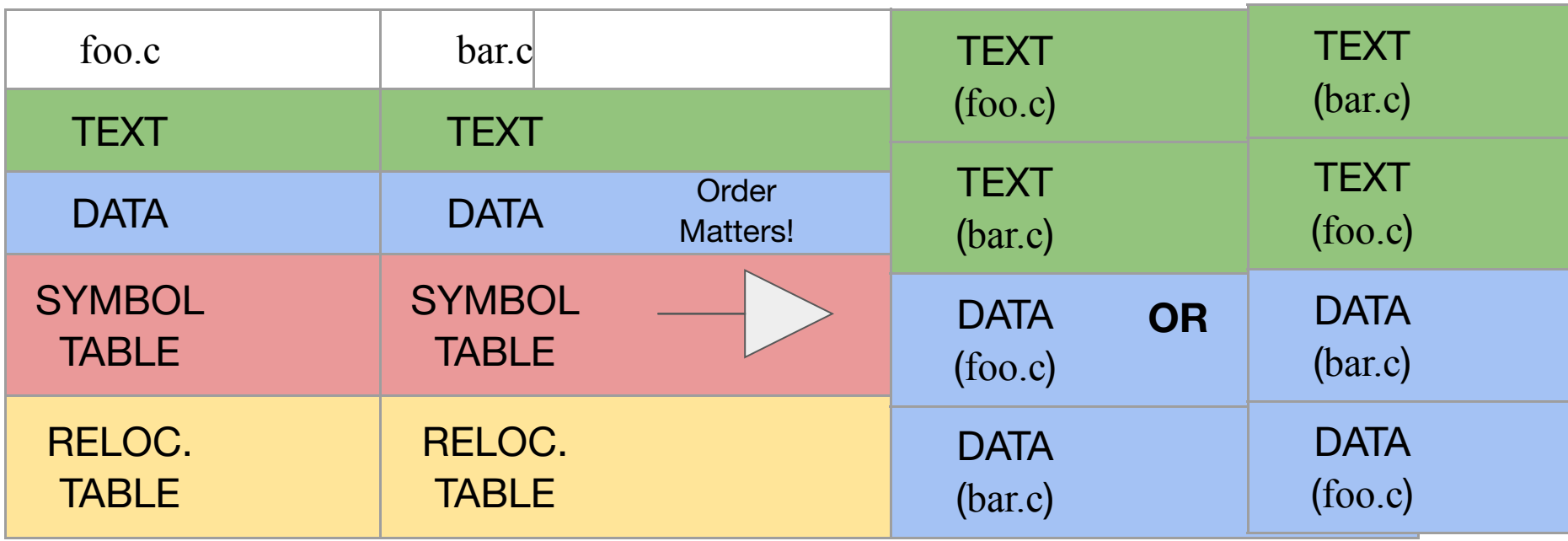
Where data goes

- All local variables go on the stack. For arguments the first 4 go into registers. The rest go on stack.
- Dynamically allocated memory goes on the heap.
- Global, static variables, and constant strings go into data segment

Executable and Linkable Format (ELF) Layout

Header	Metadata like the size of each section
Text	Machine code / Instructions
Data	Global variables, Static data (In reality, multiple sections)
Symbol Table	Mapping of symbols to address and references to symbols
Relocation Table	References to any address that may shift addresses
Debug Info	Source code to binary matching (line nums, var. names, etc.)

How Linking the Code Produces an Executable

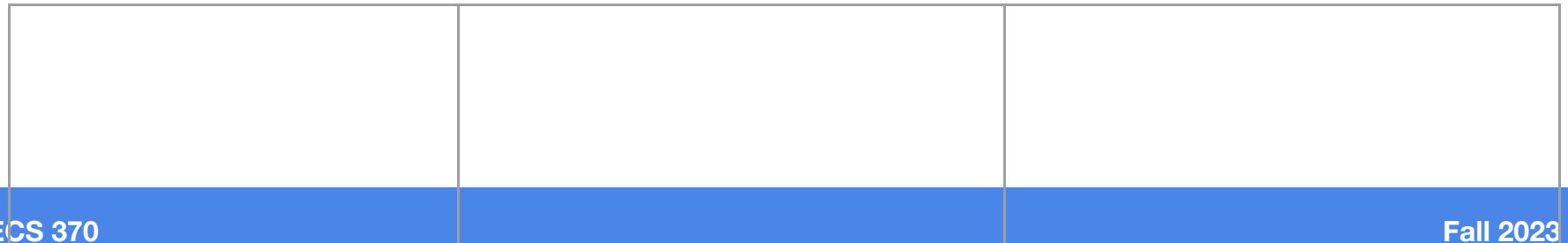


The Symbol Table Keeps Track of Globals / Statics

The **Symbol Table** tracks the locations of many things, including:

- Global variables and exported functions (global functions)
- Unresolved variables / functions (references to globals in other files)
- Static variables (Things in the data section)

Each **Symbol Table Entry** contains three key pieces of information:



The Relocation Table Tracks Changes in Address

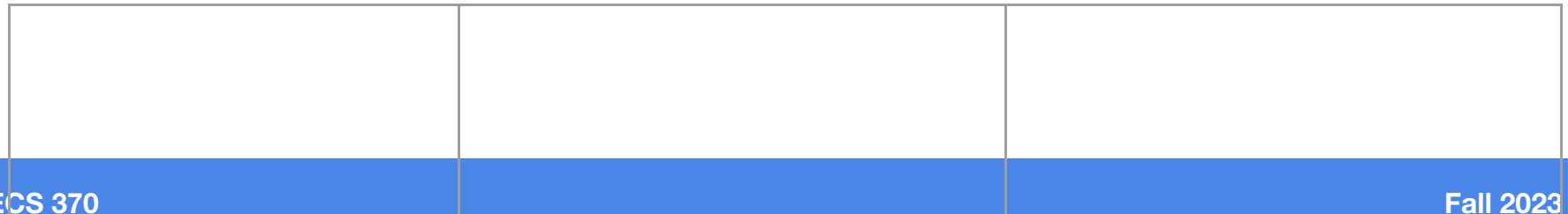
When linking files, sections in the code move around

The **Relocation Table** tracks what *instructions* need to be updated

These include:

- Calls to functions, both global and local
- References to anything in the Data section

A **Relocation Table Entry** contains the following three things:



Problem: Find the Symbol and Relocation Tables

main.c

```
1   extern int foo(int bar);
2
3   int64_t GLOBAL_VAR = 0xBEEF;
4
5   int main() {
6     int i = 0;
7
8     i = foo(i);
9     i = foo(i);
10
11    i = i + GLOBAL_VAR;
12
13    int x = rand();
14  }
15
16
```

foo.c

```
1   int bar(int value);
2
3   int foo(int value) {
4     static int FOO_STATIC = 0;
5
6     int foo_dynamic = value;
7
8     FOO_STATIC = FOO_STATIC +
9                  foo_dynamic;
10
11    return bar(FOO_STATIC);
12  }
13
14  int bar(int value) {
15    return value * 2;
16  }
```

Symbol Table for main.c

	main.c
1	`extern int foo(int bar);`
2	
3	`int64_t GLOBAL_VAR = 0xBEEF;`
4	
5	`int main() {`
6	`  int i = 0;`
7	
8	`  i = foo(i);`
9	`  i = foo(i);`
10	
11	`  i = i + GLOBAL_VAR;`
12	
13	`  int x = rand();`
14	
15	`}`
16	

Symbol Table: main.c

Label	Type	Address
foo	Undefined	Unknown
GLOBAL_VAR	Data	D + 0
main	Text	T + 0
rand	Undefined	Unknown

Symbol Table for foo.c

	foo.c
1	`int bar(int value);`
2	
3	`int foo(int value) {`
4	`  static int FOO_STATIC = 0;`
5	
6	`  int foo_dynamic = value;`
7	
8	`  FOO_STATIC = FOO_STATIC +`
9	`  foo_dynamic;`
10	
11	`  return bar(FOO_STATIC);`
12	`}`
13	
14	`int bar(int value) {`
15	`  return value * 2;`
16	`}`

Symbol Table: foo.c		
Label	**Type**	**Address**
foo	Text	T + 0
FOO_STATIC	Data	D + 0
bar	Text	T + len(foo)

Relocation Table for main.c

	main.c
1	`extern int foo(int bar);`
2	
3	`int64_t GLOBAL_VAR = 0xBEEF;`
4	
5	`int main() {`
6	`  int i = 0;`
7	
8	`  i = foo(i);`
9	`  i = foo(i);`
10	
11	`  i = i + GLOBAL_VAR;`
12	
13	`  int x = rand();`
14	`}`
15	
16	

Relocation Table: main.c

Line	Inst. Type	Dependency
9	Branch	foo
10	Branch	foo
12	Load	GLOBAL_VAR
14	Branch	rand

Relocation Table for foo.c

	foo.c
1	`int bar(int value);`
2	
3	`int foo(int value) {`
4	`  static int FOO_STATIC = 0;`
5	
6	`  int foo_dynamic = value;`
7	
8	`  FOO_STATIC = FOO_STATIC +`
9	`  foo_dynamic;`
10	
11	`  return bar(FOO_STATIC);`
12	`}`
13	
14	`int bar(int value) {`
15	`  return value * 2;`
16	`}`

Relocation Table: foo.c		
Line	**Inst. Type**	**Dependency**
8	Load	FOO_STATIC
8	Store	FOO_STATIC
11	Load	FOO_STATIC
11	Branch	bar

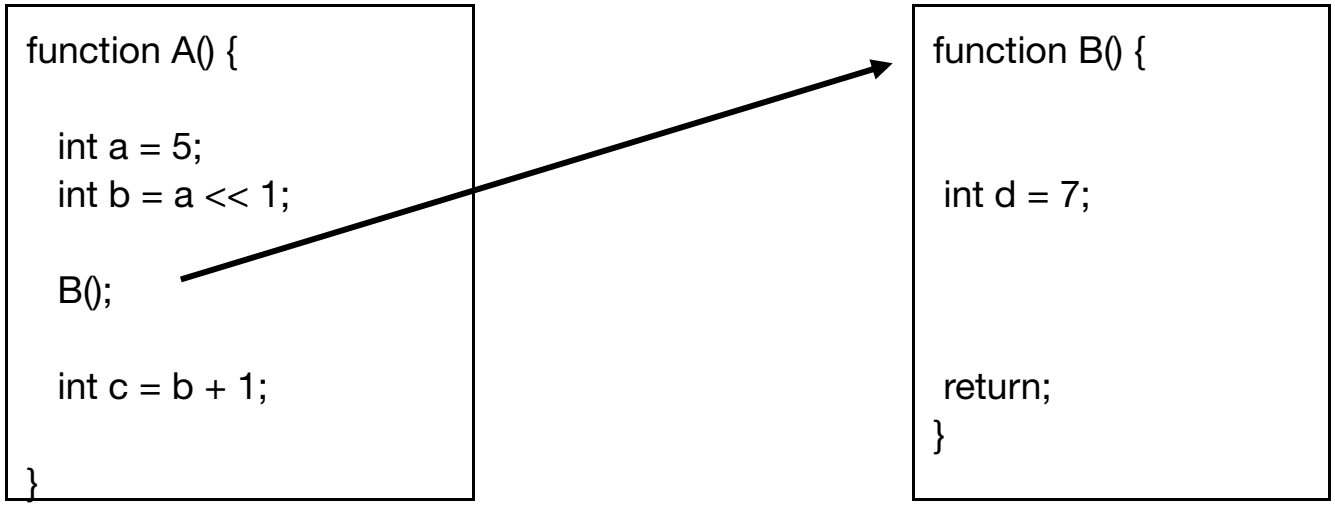
Topic Covered

- Understanding the LC2K ISA
- ARM
- Endianness
- Linking/Linker
- **Caller/Callee**
- LC2K Processor Performance
  - Single-Cycle Datapath
  - Multi-Cycle Datapath
- Pipeline Datapath
- CheatSheet Stuff

Caller/Callee Save

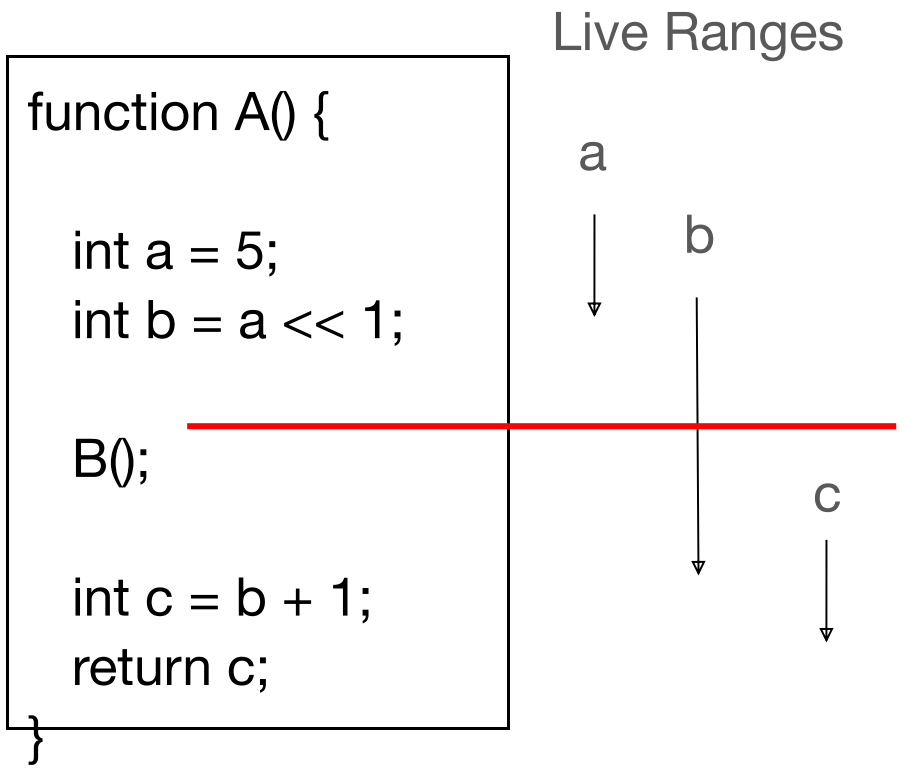
- Caller-saved: Don't want the function we call to mess up our registers

A doesn't want its registers to be overwritten by B. Save registers before call, load after call.



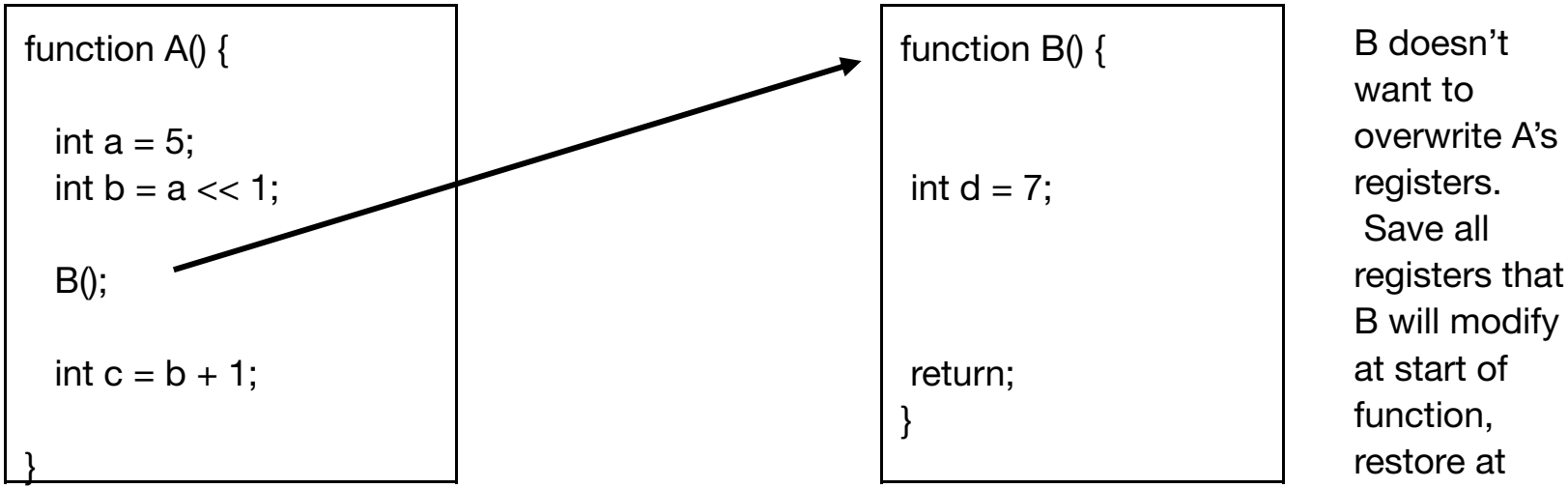
Caller/Callee Save

- A variable gets **caller-saved** when its function calls another function and the variable is live across the function call
  - A variable is *live* if the value of the variable is needed after the function call.
  - Liveness is the only optimization we make for caller/callee saving



Caller/Callee Save

- Callee-saved: Don't want to overwritten the registers of the function that called us



B doesn't want to overwrite A's registers. Save all registers that B will modify at start of function, restore at end.

Caller/Callee Save

- A variable gets **callee-saved** every time its function gets called
  - The **main** function does not need to callee save its variables.
  - Note: A function is not main function just because caller is not shown

No function calls main, no callee saves needed.

```
function main() {

   int x = 100;

   ...

}
```

B can be called by other functions, callee saves needed.

```
function B() {


 int d = 7;



 return;
}
```

Caller/Callee

How many STUR/LDUR pairs are needed for each variable assuming all registers are caller-saved?

```
int myFunc(int x) {
    int y, z;
    y = x;
    printf("Hello");
        return y;
}
```

```
int main() {
  int a, b, c;
  a = 0; b = 1;
  for (int i = 0; i < 3; i++){
     c = myFunc(b);
     a += c;
        }
        b = a;
        return c;
}
```

Caller/Callee

How many STUR/LDUR pairs are needed for each variable assuming all registers are caller-saved?

```
int myFunc(int x) {
  int y, z;
  y = x;
  printf("Hello");
        return y;
}
```

```
int main() {
  int a, b, c;
  a = 0; b = 1;
  for (int i = 0; i < 3; i++){
    c = myFunc(b);
    a += c;
      }
      b = a;
      return c;
}
```

y	3/3	a	3/3
x	0/0	b	3/3
z	0/0	c	0/0
	i	3/3	

Caller/Callee

How many STUR/LDUR pairs are needed for each variable assuming all registers are callee-saved?

```
int myFunc(int x) {
  int y, z;
  y = x;
  printf("Hello");
        return y;
}
```

```
int main() {
  int a, b, c;
  a = 0; b = 1;
  for (int i = 0; i < 3; i++){
     c = myFunc(b);
     a += c;
        }
        b = a;
        return c;
}
```

Caller/Callee

How many STUR/LDUR pairs are needed for each variable assuming all registers are callee-saved?

```
int myFunc(int x) {
  int y, z;
  y = x;
  printf("Hello");
        return y;
}
```

```
int main() {
  int a, b, c;
  a = 0; b = 1;
  for (int i = 0; i < 3; i++){
    c = myFunc(b);
    a += c;
      }
      b = a;
      return c;
}
```

y	3/3	a	0/0
x	3/3	b	0/0
z	3/3	c	0/0
	i	0/0	

Caller/Callee

How many STUR/LDUR pairs are needed assuming 2 of each type of register?

caller loads/stores

y	3/3	a	3/3
x	0/0	b	3/3
z	0/0	c	0/0
	i	3/3	

callee loads/stores

y	3/3	a	0/0
x	3/3	b	0/0
z	3/3	c	0/0
	i	0/0	

```
int myFunc(int x) {
  int y, z;
  y = x;
  printf("Hello");
        return y;
}

int main() {
  int a, b, c;
  a = 0; b = 1;
  for (int i = 0; i < 3; i++){
     c = myFunc(b);
     a += c;
        }
        b = a;
        return c;
}
```

Caller/Callee

How many STUR/LDUR pairs are needed assuming 2 of each type of register?

caller loads/stores

y	3/3	a	3/3
x	0/0	b	3/3
z	0/0	c	0/0
	i	3/3	

callee loads/stores

y	3/3	a	0/0
x	3/3	b	0/0
z	3/3	c	0/0
	i	0/0	

2 registers of each type

y	3/3 (callee)	a	0/0 (callee)
x	0/0 (caller)	b	0/0 (callee)
z	0/0 (caller)	c	0/0 (caller)
	i	3/3 (caller)	

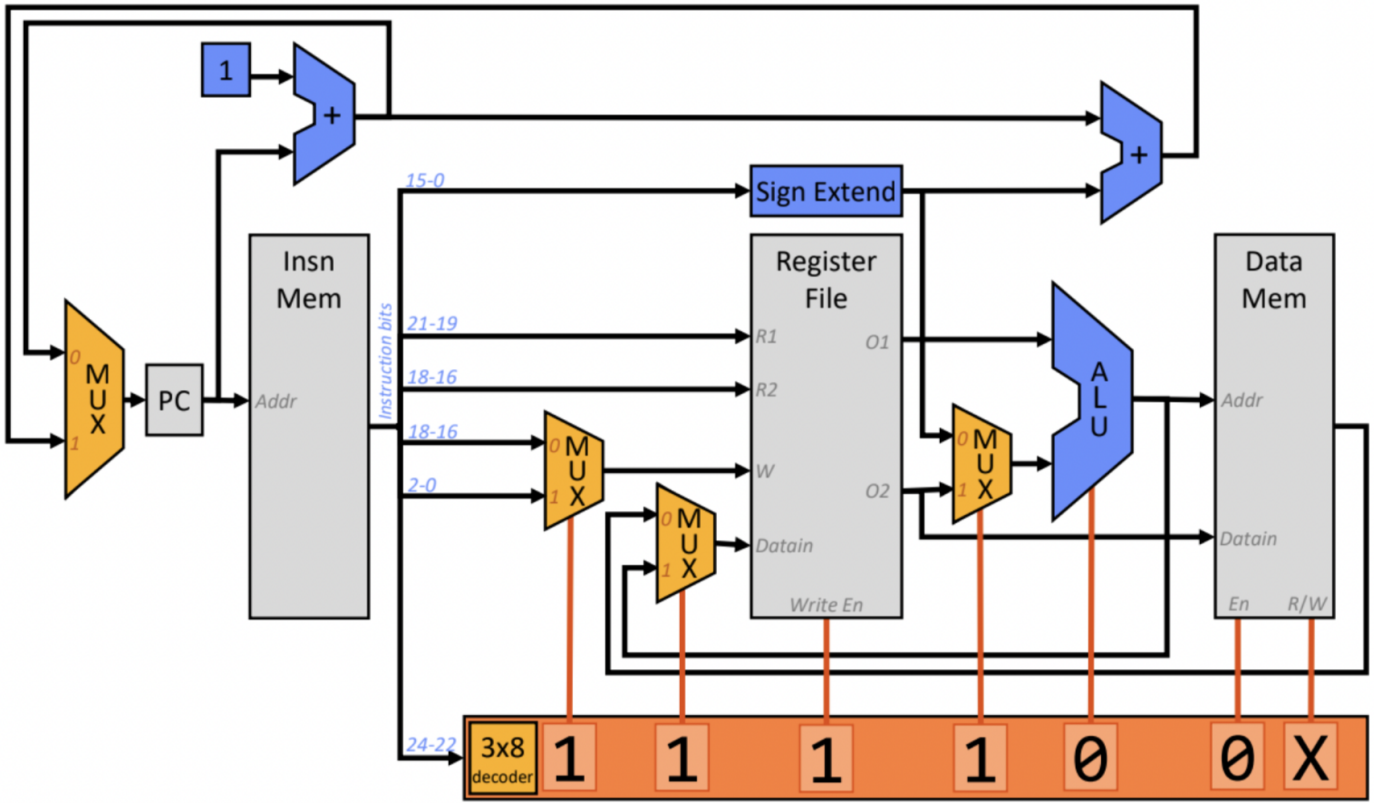
```
int myFunc(int x) {
  int y, z;
  y = x;
  printf("Hello");
        return y;
}

int main() {
  int a, b, c;
  a = 0; b = 1;
  for (int i = 0; i < 3; i++){
     c = myFunc(b);
     a += c;
        }
        b = a;
        return c;
}
```

Topics Covered

- Understanding the LC2K ISA
- ARM
- Endianness
- Linking/Linker
- Caller/Callee
- LC2K Processor Performance
  - **Single-Cycle Datapath**
  - Multi-Cycle Datapath
- Pipeline Datapath
- CheatSheet Stuff

Single Cycle Datapath

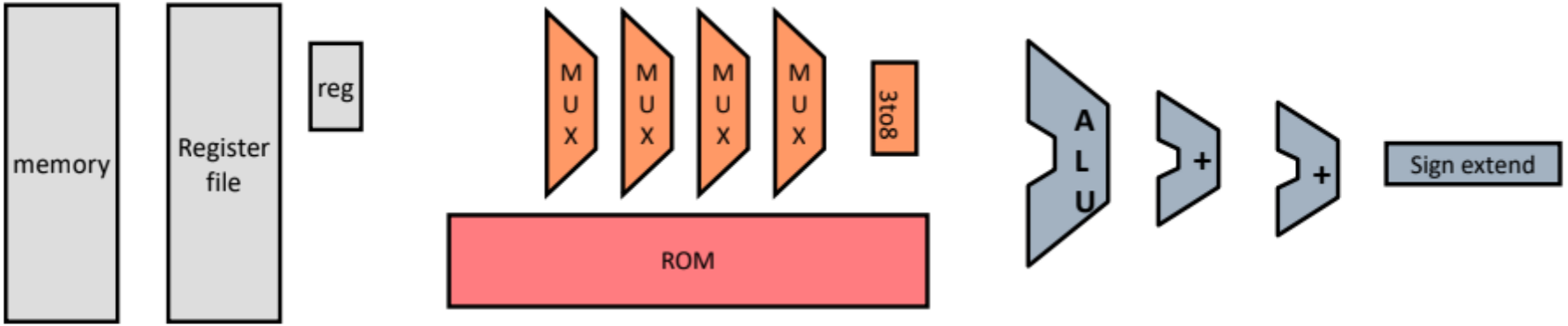
Hardware to run instructions



Datapath Uses Components That Work Together

We *could* wire components together to produce output for a specific instruction, but then we would have 8+ individual circuits

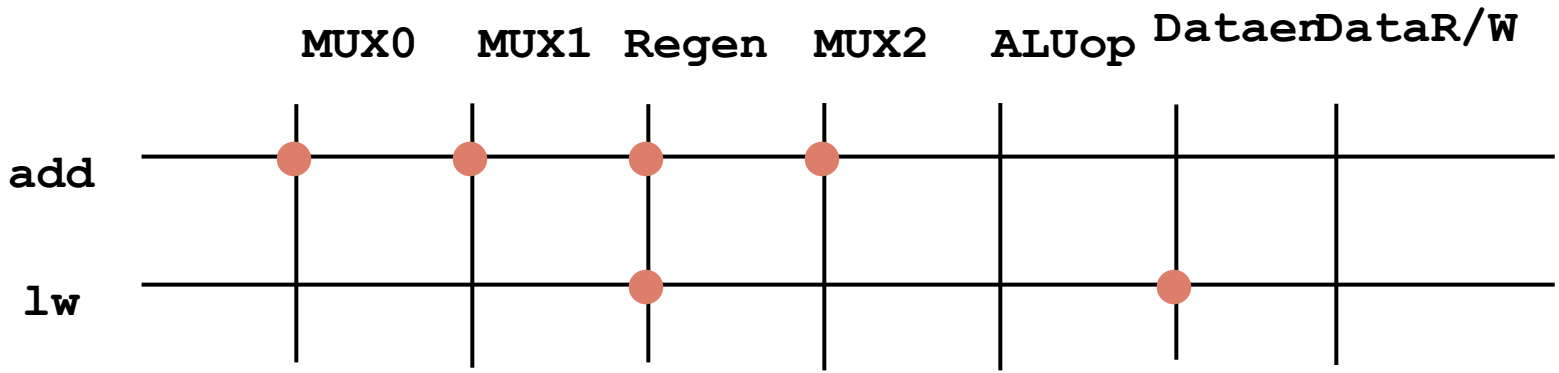
Instead, connect all possible inputs that an instruction could execute and use a MUX to select inputs for specific instructions



Use a ROM to Select Inputs in Our MUXes

The ROM specifies which inputs and outputs each component uses

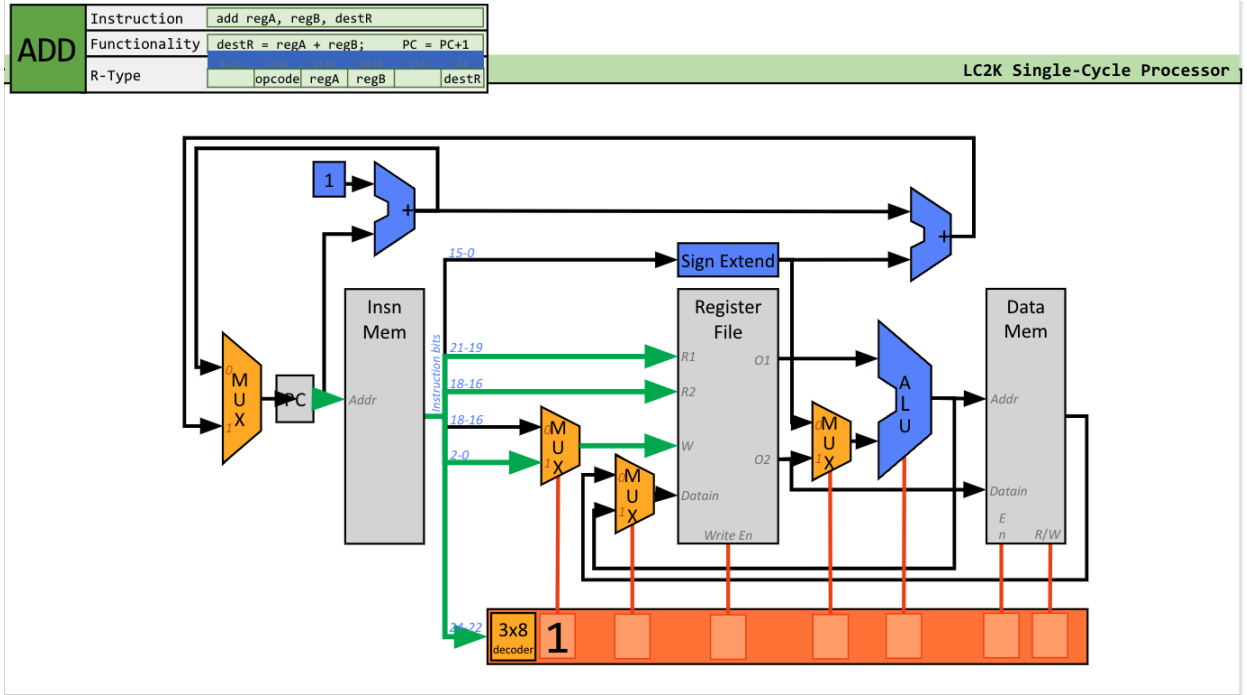
Each entry in the ROM determines the behavior of one opcode



add 1 2 3

reg3 = reg1 + reg2

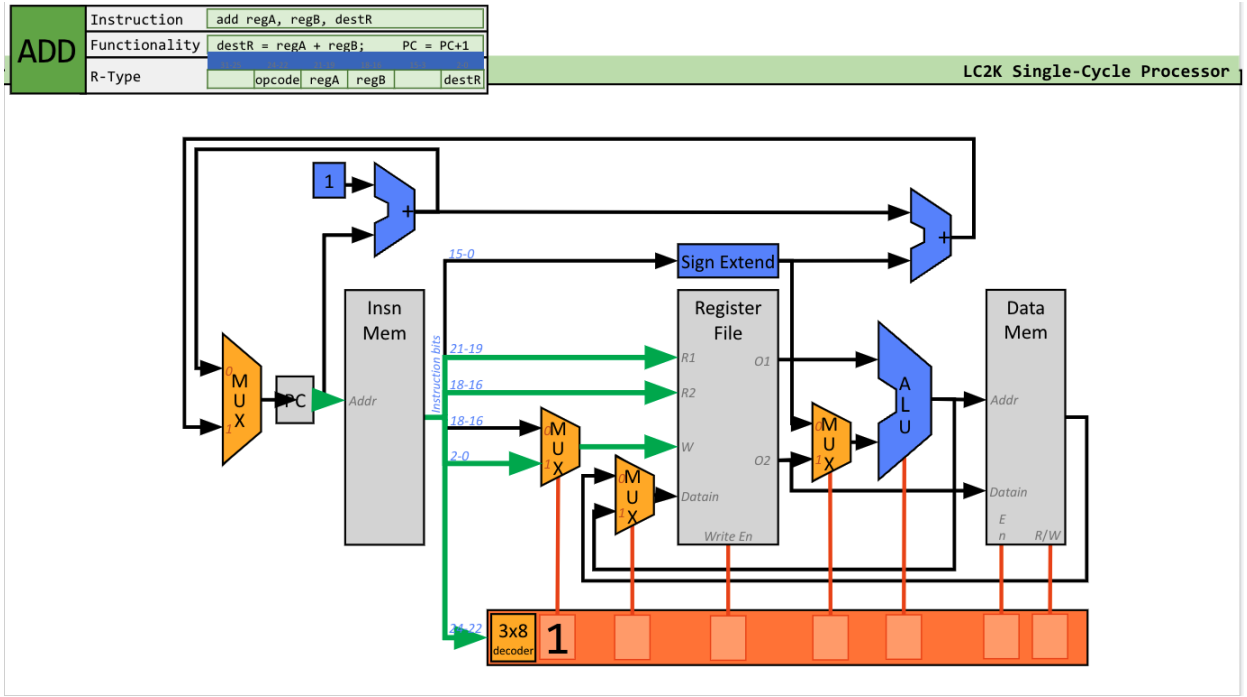
. Fetch the instruction and PC+1



add 1 2 3

reg3 = reg1 + reg2

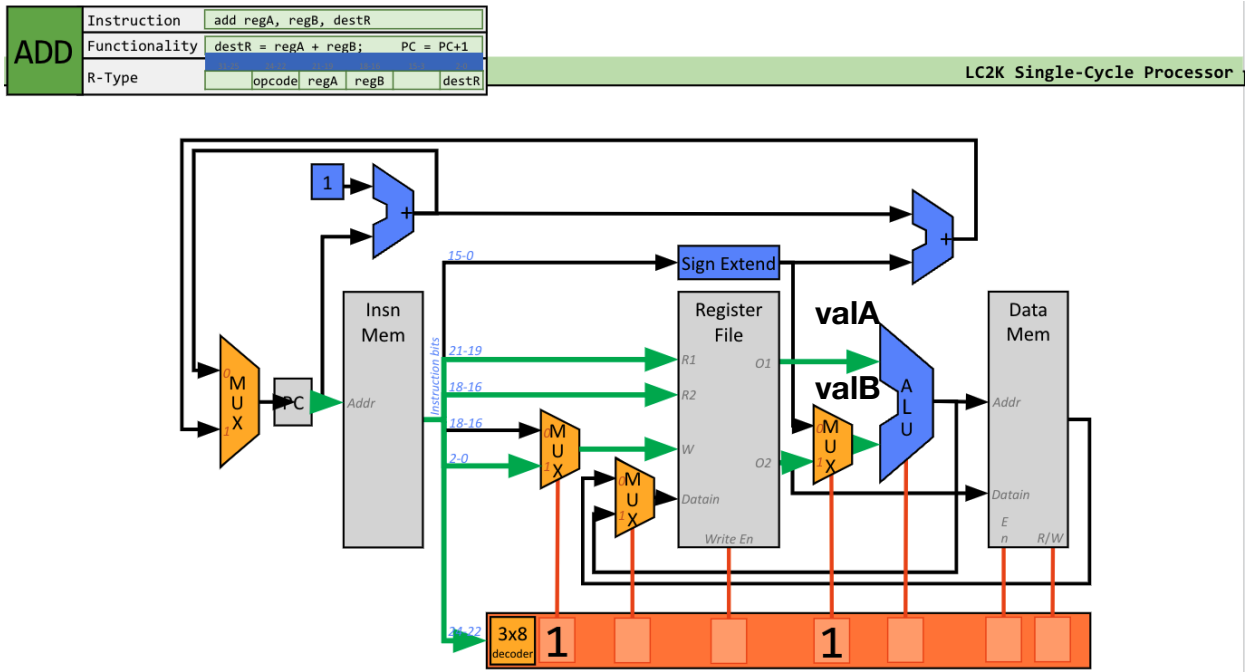
- Fetch the instruction and PC+1
- Decode and set control bits



add 1 2 3

reg3 = reg1 + reg2

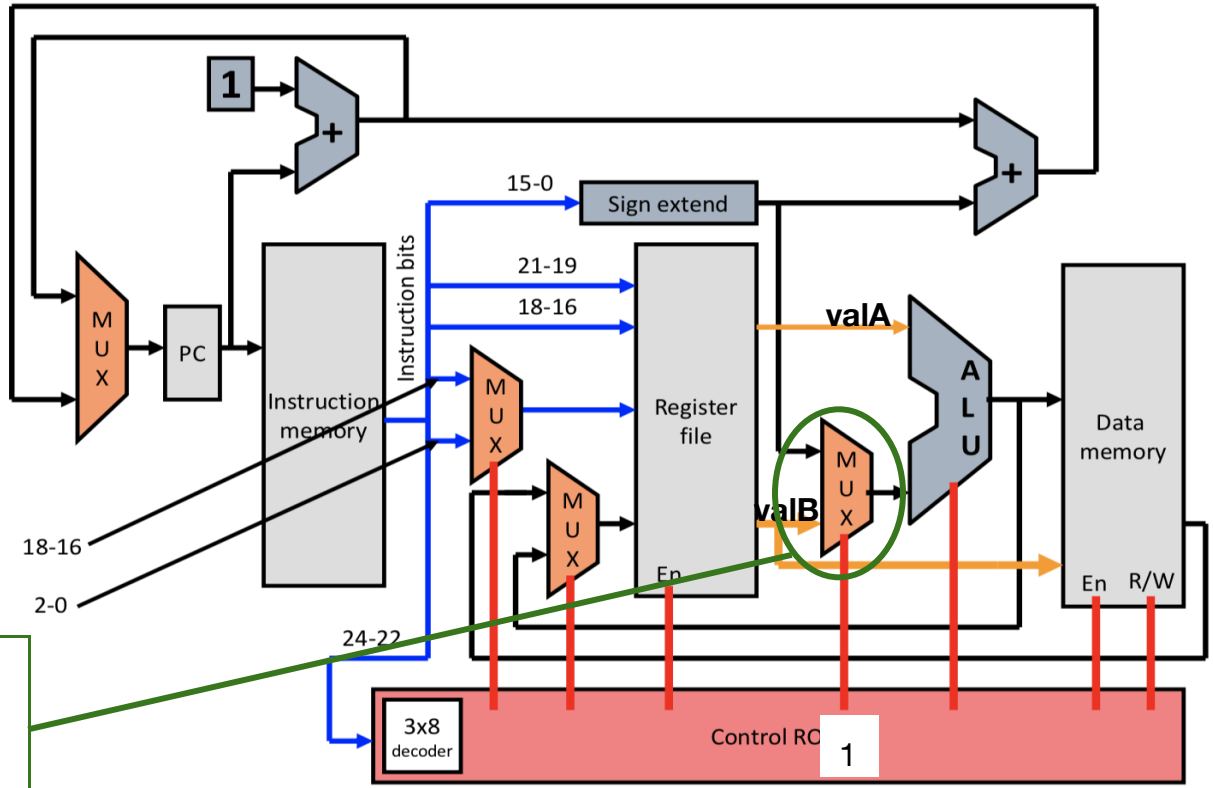
- Fetch the instruction and PC+1
- Decode and set control bits
- Read the register value



add 1 2 3

reg3 = reg1 + reg2

1. Fetch the instruction and PC+1
2. Decode and set control bits
3. Read the register value
4. Do ALU operation

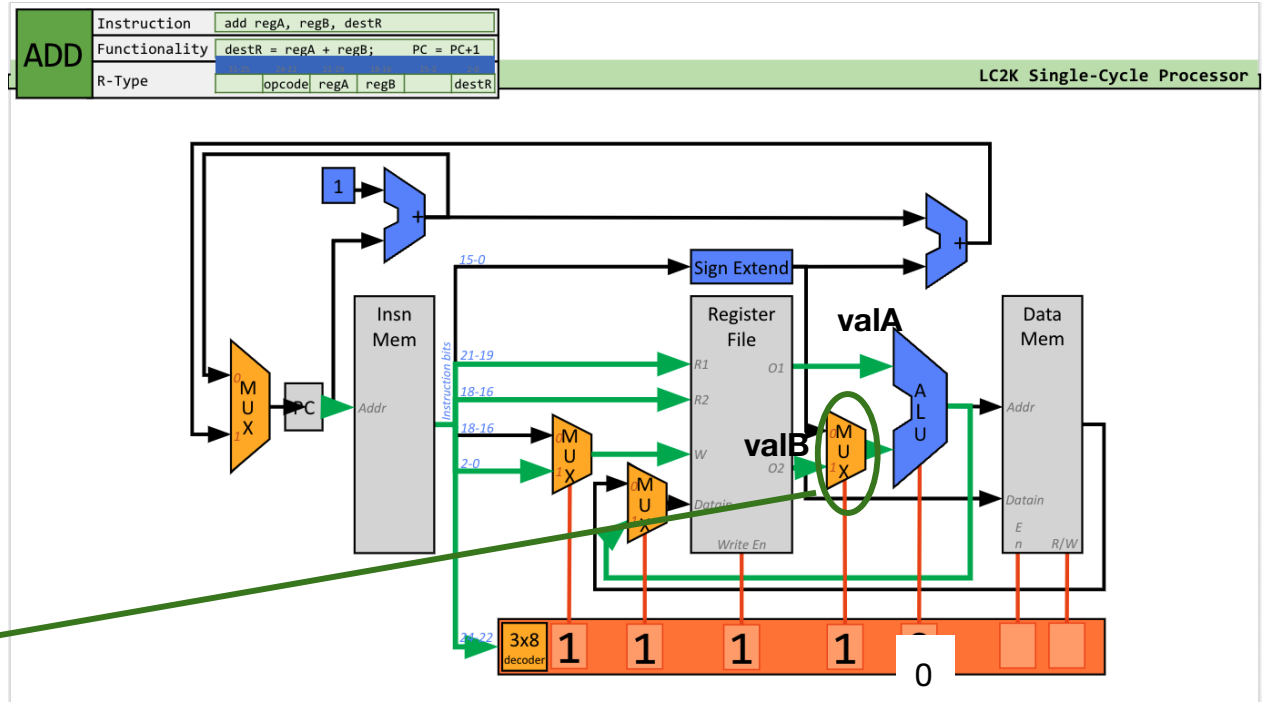


Select input for ALU
S=0 Offset (lw, sw)
S=1 Value in regB (add, nor, beq)

add 1 2 3

reg3 = reg1 + reg2

- Fetch the instruction and PC+1
- Decode and set control bits
- Read the register value
- Do ALU operation

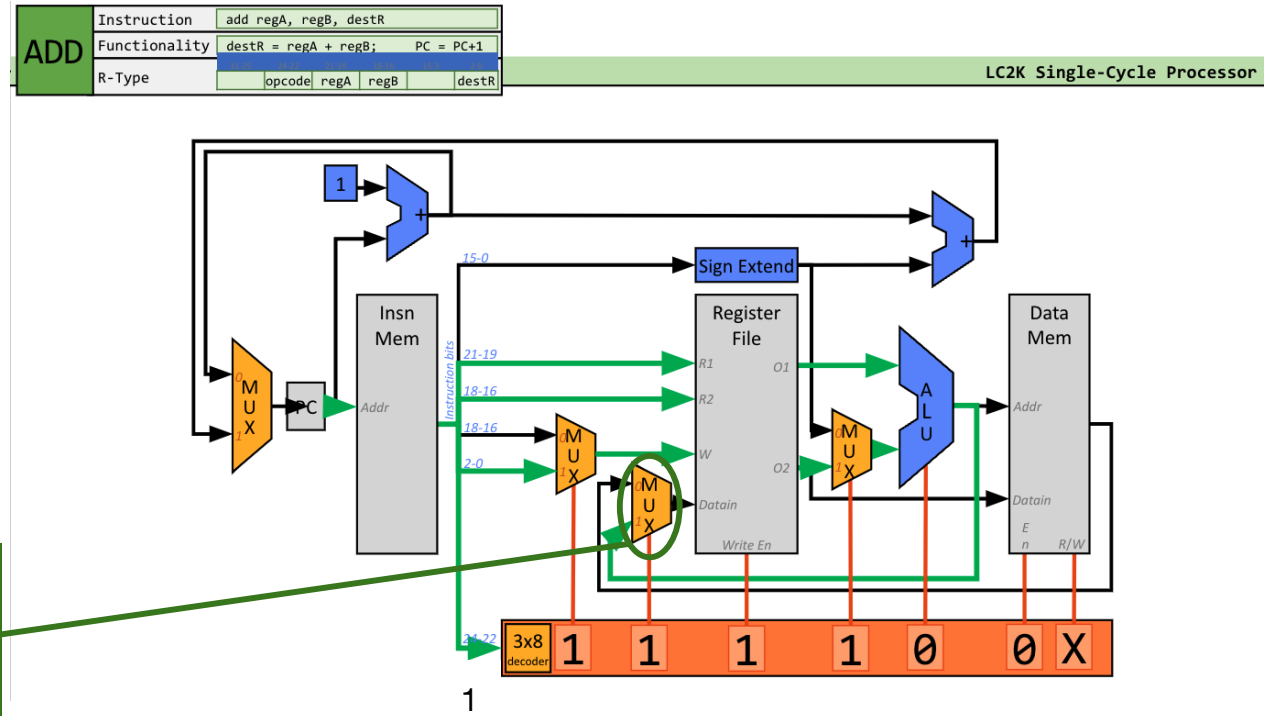


ALU operations
S=0 add operation (add, lw, sw)
S=1 nor operation (nor)
Compare (beq)

add 1 2 3

reg3 = reg1 + reg2

1. Fetch the instruction and PC+1
2. Decode and set control bits
3. Read the register value
4. Do ALU operation
5. Writeback to register file

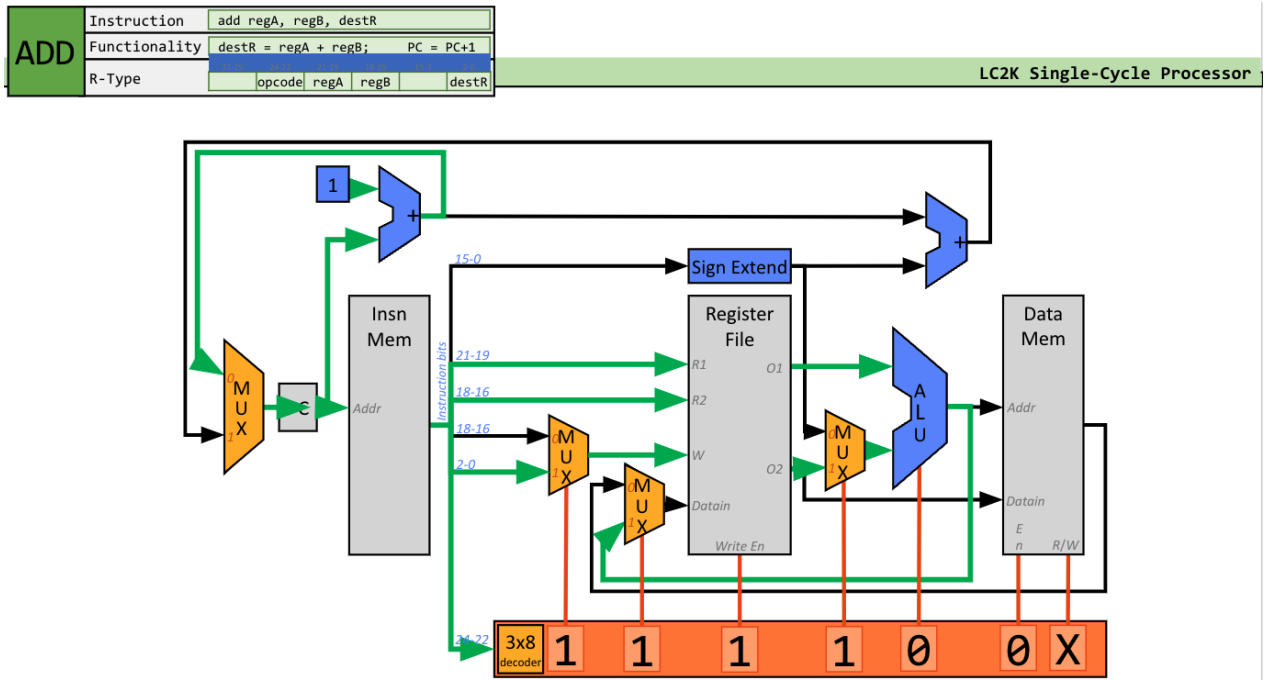


Select write reg
S=0 18-16 bit (lw, jalr)
S=1 0-2 bit (add, nor)

add 1 2 3

reg3 = reg1 + reg2

- Fetch the instruction and PC+1
- Decode and set control bits
- Read the register value
- Do ALU operation
- Writeback to register file

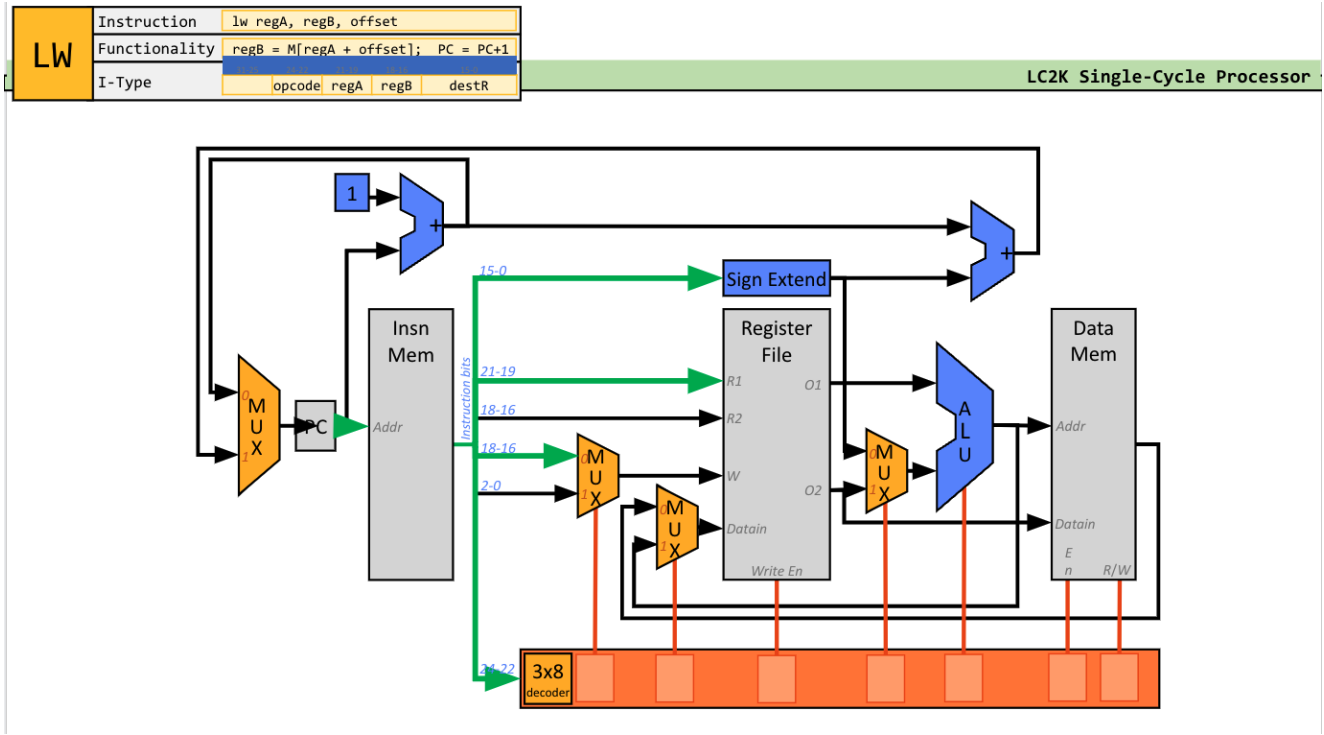


1

lw 0 2 3

reg2 = Mem[reg0 + 3]

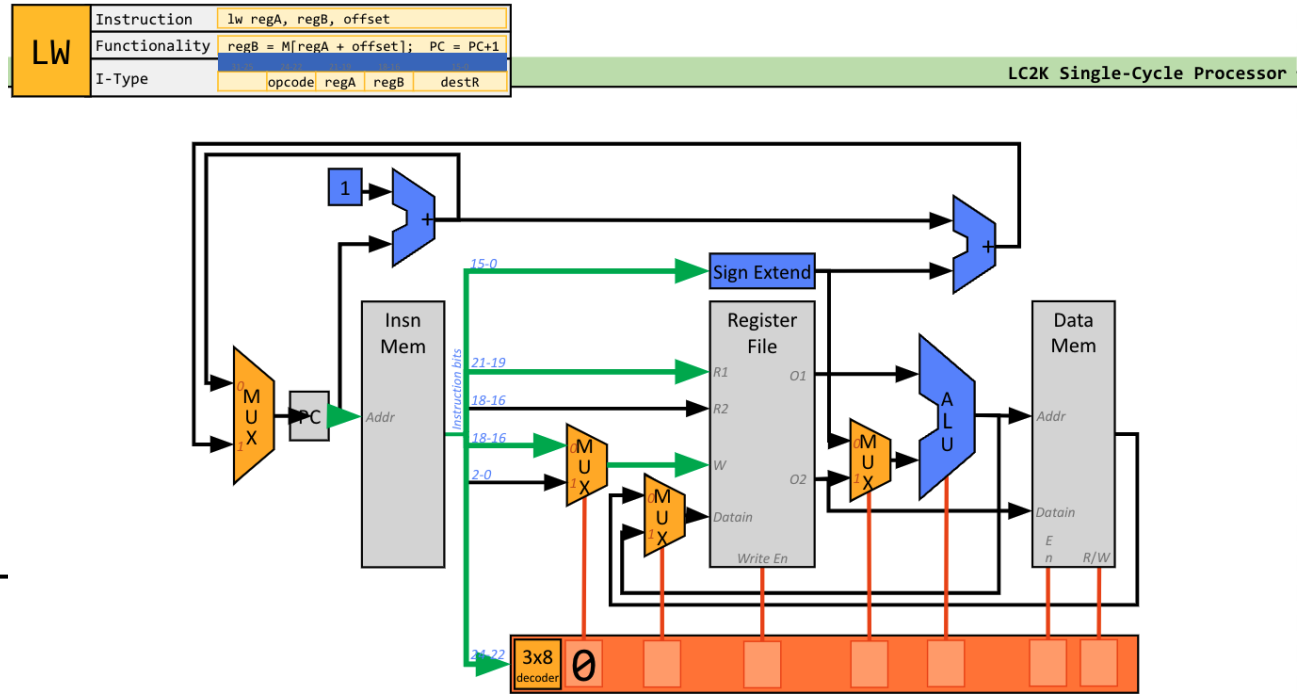
. Fetch the instruction and PC+1



lw 0 2 3

reg2 = Mem[reg0 + 3]

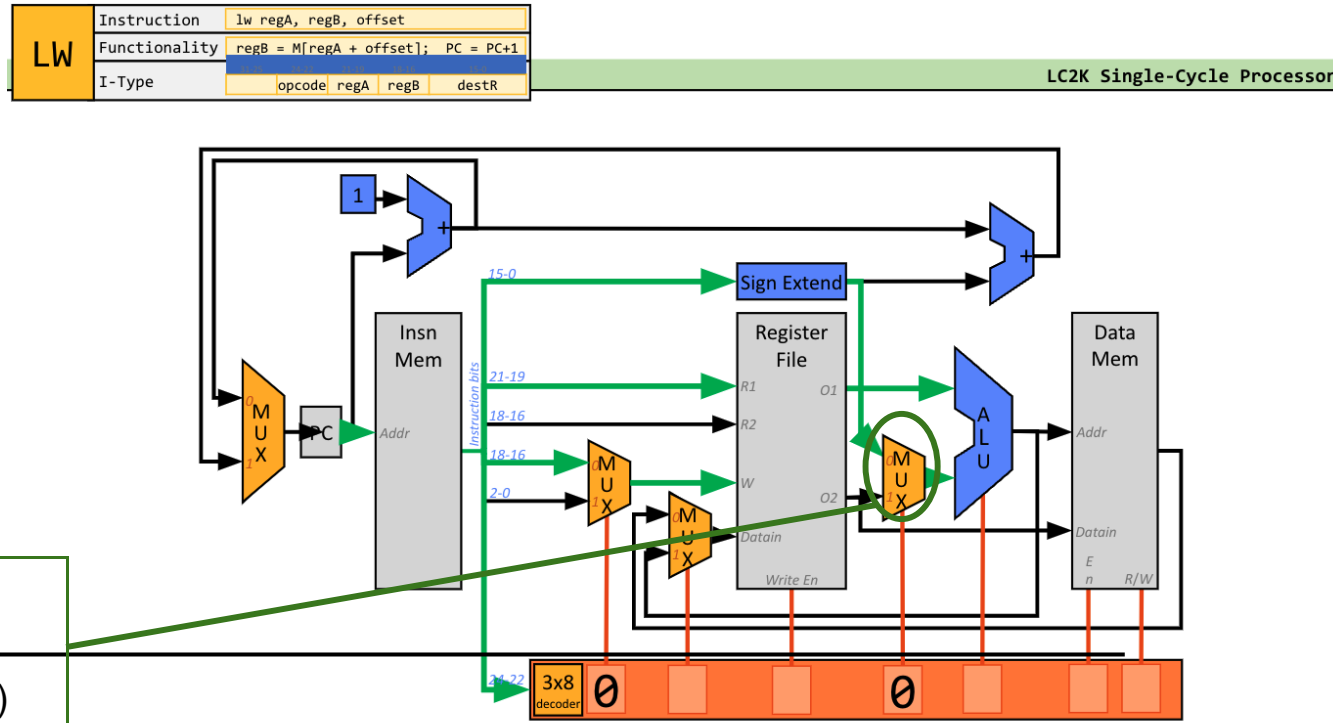
- Fetch the instruction and PC+1
- Decode and set control bits



lw 0 2 3

reg2 = Mem[reg0 + 3]

1. Fetch the instruction and PC+1
2. Decode and set control bits
3. Read the register value & offset

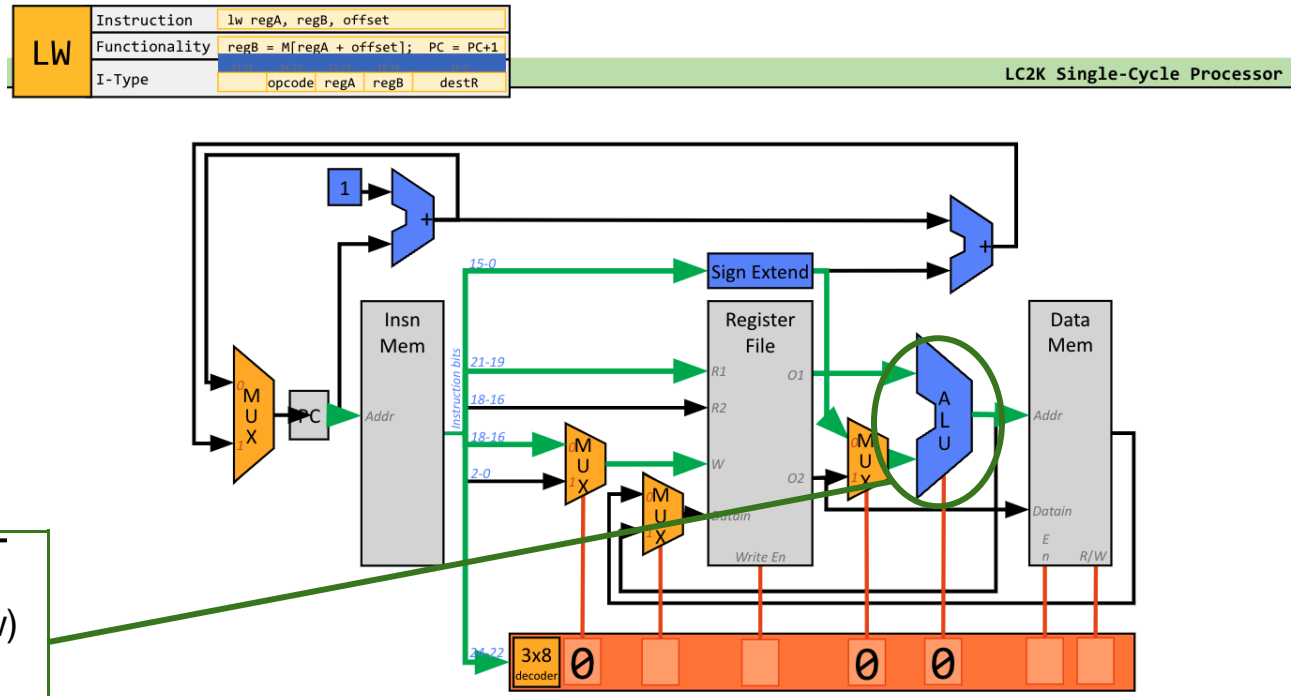


Select input for ALU
S=0 Offset (lw, ~~sw~~)
S=1 Value in regB (add, nor, beq)

lw 0 2 3

reg2 = Mem[reg0 + 3]

- Fetch the instruction and PC+1
- Decode and set control bits
- Read the register value
- Do ALU operation



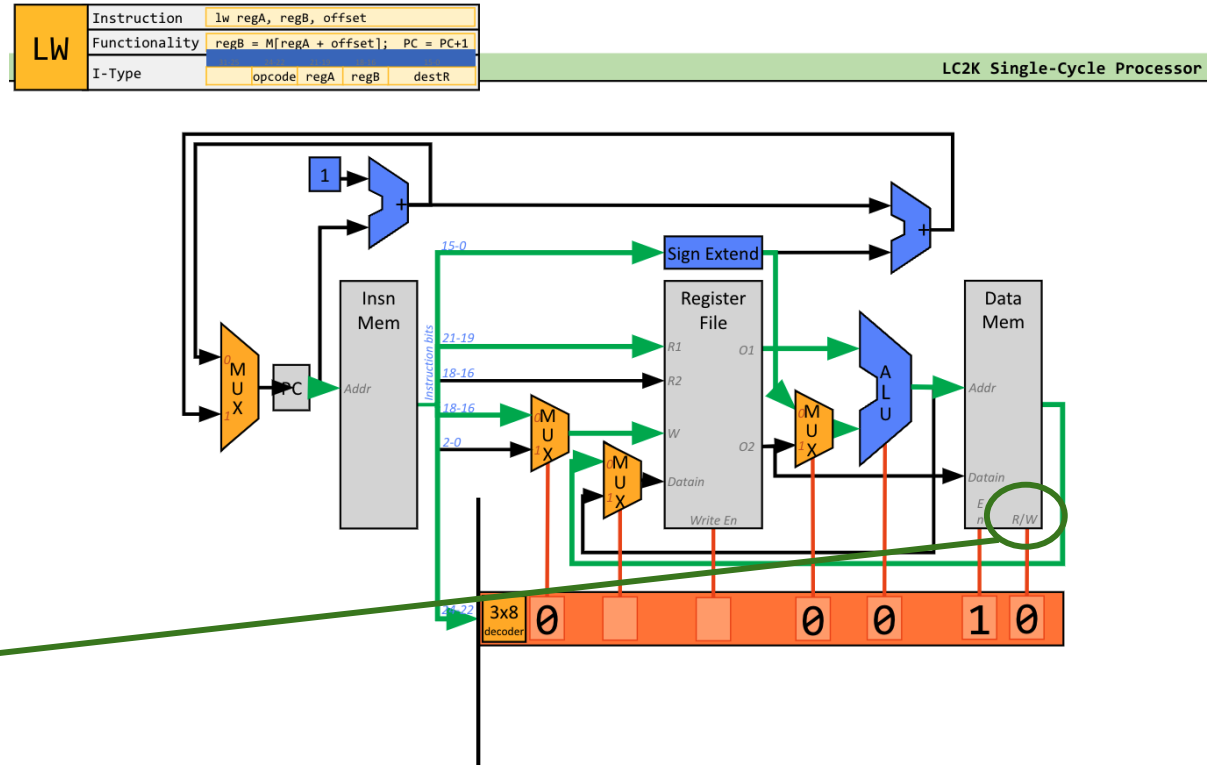
ALU operations
S=0 add operation (add, lw, sw)
S=1 nor operation (nor)
Compare (beq)

lw 0 2 3

reg2 = Mem[reg0 + 3]

- Fetch the instruction and PC+1
- Decode and set control bits
- Read the register value
- Do ALU operation
- Memory Access

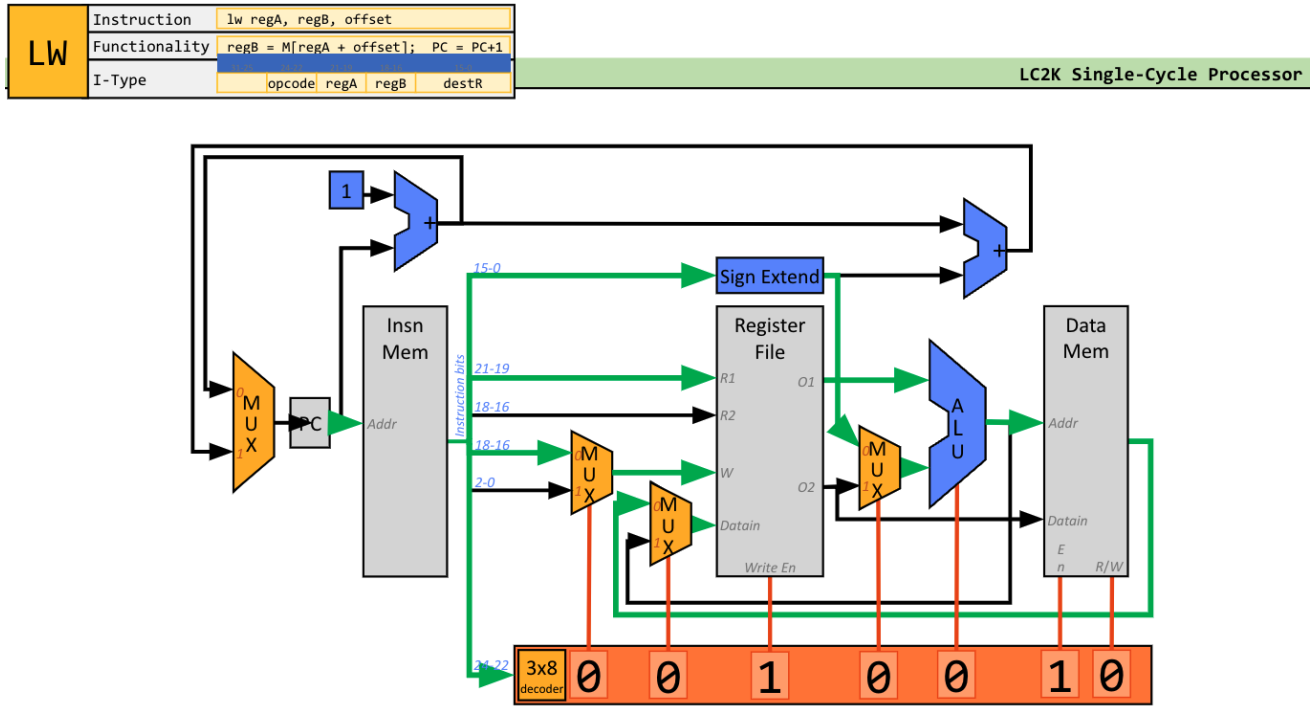
Data memory
En: access memory or not
R/W: read or write



lw 0 2 3

reg2 = Mem[reg0 + 3]

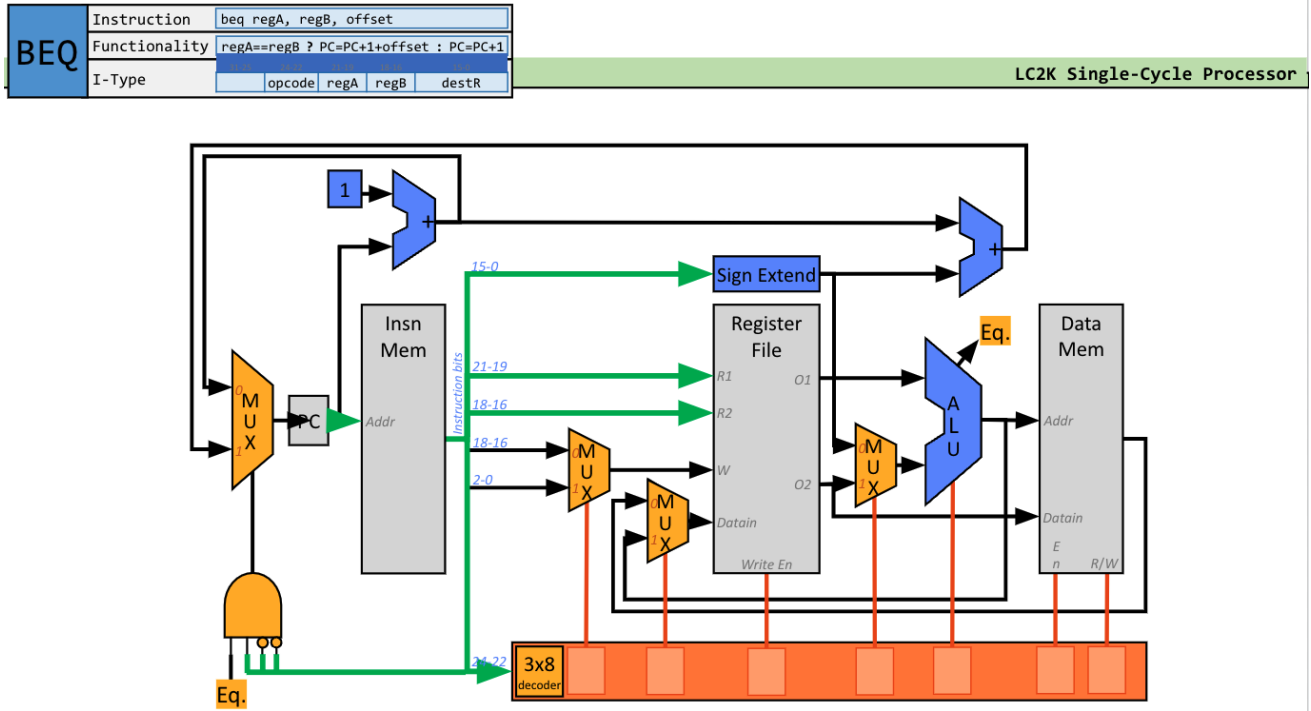
- Fetch the instruction and PC+1
- Decode and set control bits
- Read the register value
- Do ALU operation
- Memory Access
- Writeback



beq 0 1 offset

if reg0 == reg1 {pc = pc + 1 + offset}

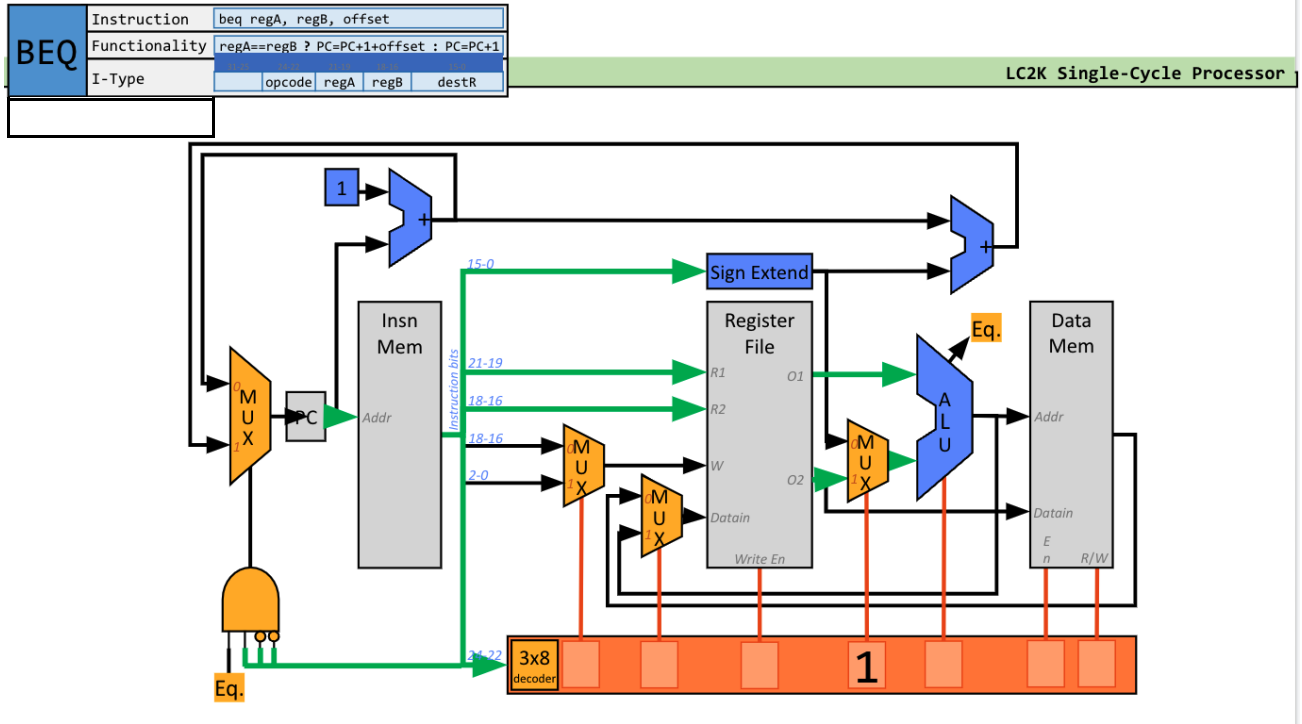
. Fetch the instruction and PC+1



beq 0 1 offset

if reg0 == reg1 {pc = pc + 1 + offset}

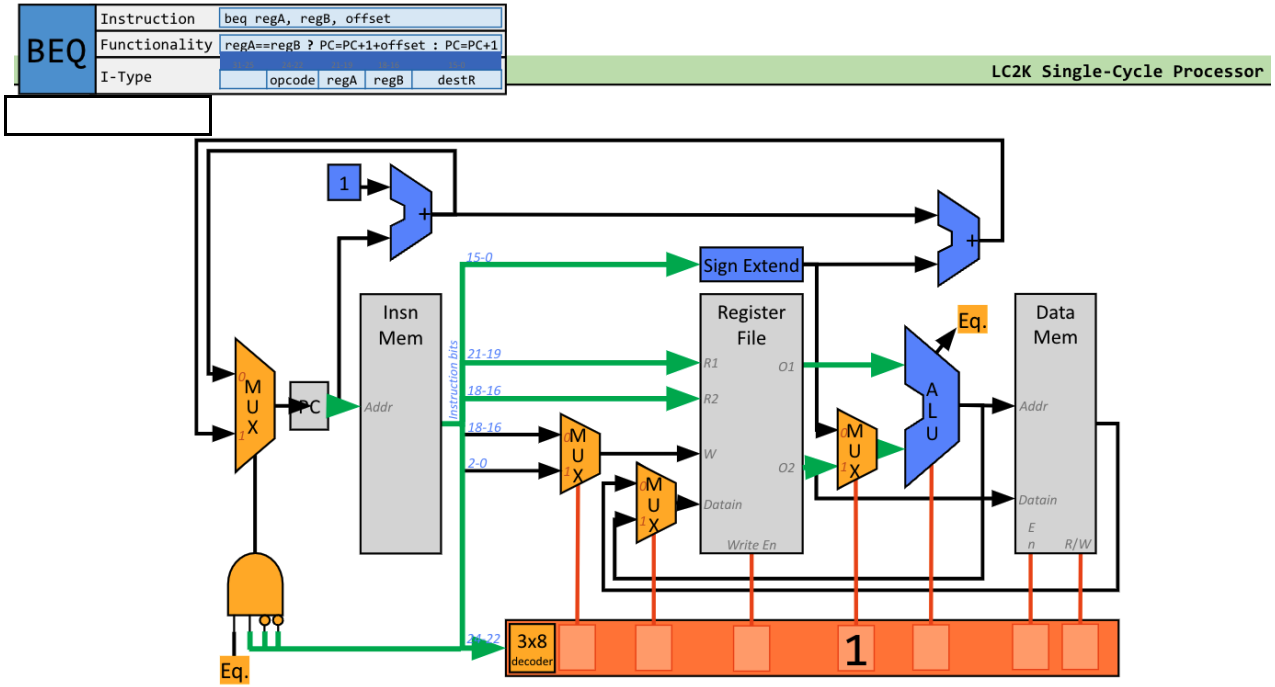
- Fetch the instruction and PC+1
- Decode and set control bits



beq 0 1 offset

if reg0 == reg1 {pc = pc + 1 + offset}

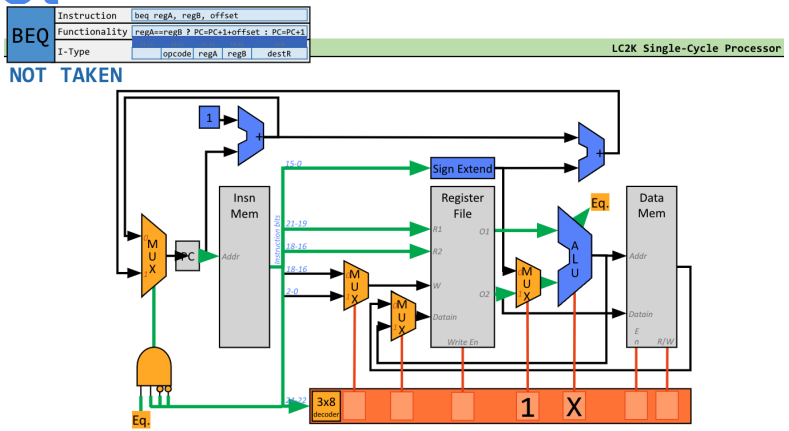
- Fetch the instruction and PC+1
- Decode and set control bits
- Read the register value

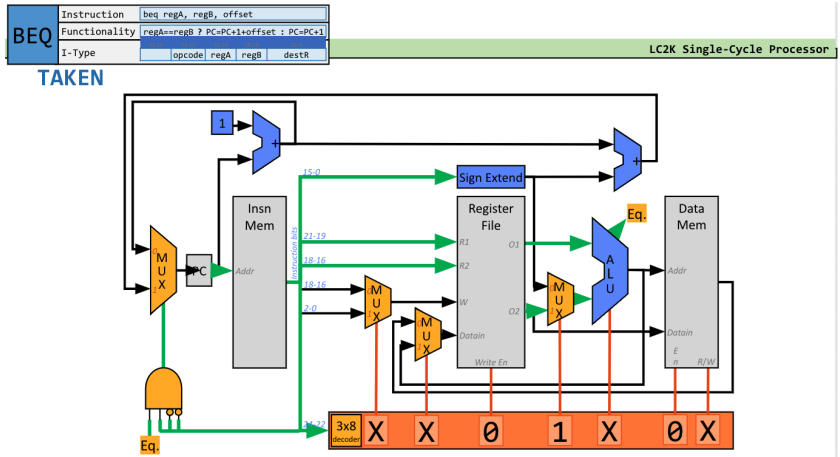


beq 0 1 offset

if reg0 == reg1 {pc = pc + 1 + offset}

- Fetch the instruction and PC+1
- Decode and set control bits
- Read the register value
- Do ALU operation and PC+1+offset





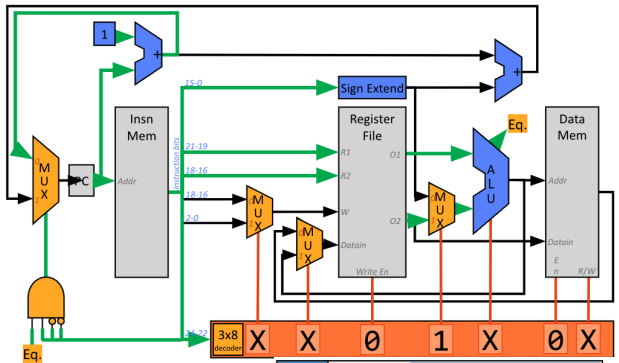
beq 0 1 offset

if reg0 == reg1 {pc = pc + 1 + offset}

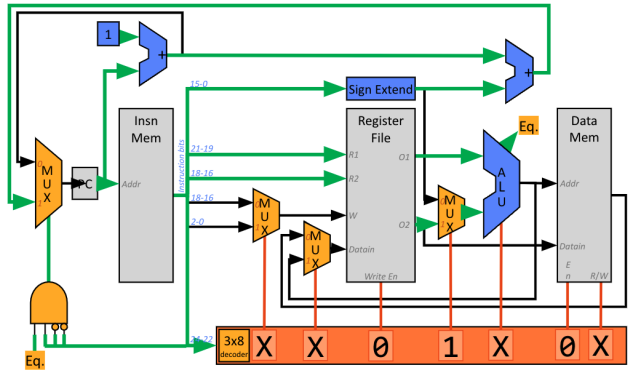
1. Fetch the instruction and PC+1
2. Decode and set control bits
3. Read the register value
4. Do ALU operation and PC+1+offset
5. Select next PC

Next PC
S=0 PC+1
S=1 PC+1+offset

NOT TAKEN



TAKEN



Single Cycle Datapath

For each instruction

- Think about what steps needed to execute the instruction
- For each step, find the component needed
- Find the input and output for that component
- Hint for cheatsheet:
  - Label a single cycle datapath for the different instruction types, follow through what happens in the datapath
  - Similar to what we did in the last slides

Which Components Do Instructions Use

Instr.	Instr. Mem. Access	Read Reg.	ALU	Data Mem. Access	Write Reg.
`add`	✓	✓	✓		✓
`nor`	✓	✓	✓		✓
`lw`	✓	✓	✓	✓	✓
`sw`	✓	✓	✓	✓	
`beq`	✓	✓	✓		
`jalr`	✓	✓			✓
`noop`	✓				
`halt`	✓				

*NOTE: This table **DOES NOT** correspond directly with the number of cycles required per instruction for the multicycle datapath

Single Cycle Datapath

Performance

- CPI (cycle per instruction) will always be 1
- Clock period depends on slowest **instruction** (lw or sw usually)

Problem 2a: Single-Cycle Datapath Benchmarking

We are using a benchmarking program that executes 15,000 instructions. 35% are **`lw`**, 15% **`sw`**, 20% **`add`** / **`nor`**, 15% **`beq`**, and 15% **`noop`** / **`halt`**.

Each use of the ALU costs **20ns**, reading memory costs **50ns**, writing to memory costs **70ns**, register file reads and writes cost **10ns**, and everything else costs **0ns**.

What is the runtime of this benchmark on our single-cycle system?

Problem 2a: Single-Cycle Datapath Benchmarking

What is the runtime of this benchmark on our single-cycle system?

Problem 2a: Single-Cycle Datapath Benchmarking

What is the runtime of this benchmark on our single-cycle system?

Clock Cycle = Slowest Instr. (`sw`) =

50ns (Instr. Fetch) + 10ns (Reg. Read) + 20ns (ALU)
+ 70ns (Data Fetch) = 150ns / Cycle

Runtime = (Time / Cycle) × (Cycles / Instr.) × (# of Instr.)
= 150ns / Cycle × 1 × 15,000 Instr.
= **2,250,000ns**

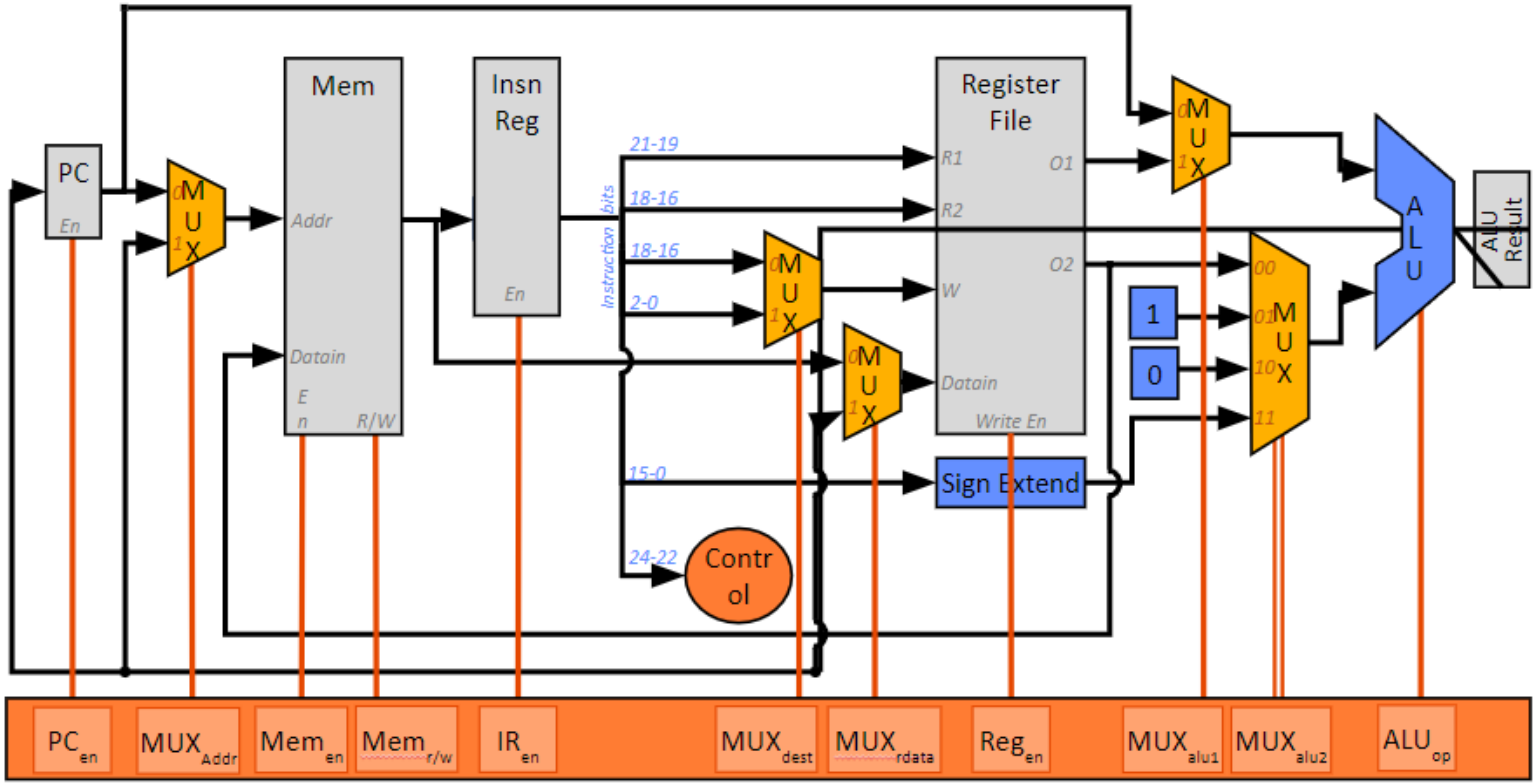
Single Cycle Drawbacks

- All instructions take the same, slowest amount of time to execute
  - Noop/halt only need to take very little time to execute, but they have to take the same time as an lw/sw
- Many components are duplicated across the datapath (memory files, multiple adders, etc)

Topic Covered

- Understanding the LC2K ISA
- ARM
- Endianness
- Linking/Linker
- Caller/Callee
- LC2K Processor Performance
  - Single-Cycle Datapath
  - **Multi-Cycle Datapath**
- Pipeline Datapath
- CheatSheet Stuff

Multi Cycle Datapath

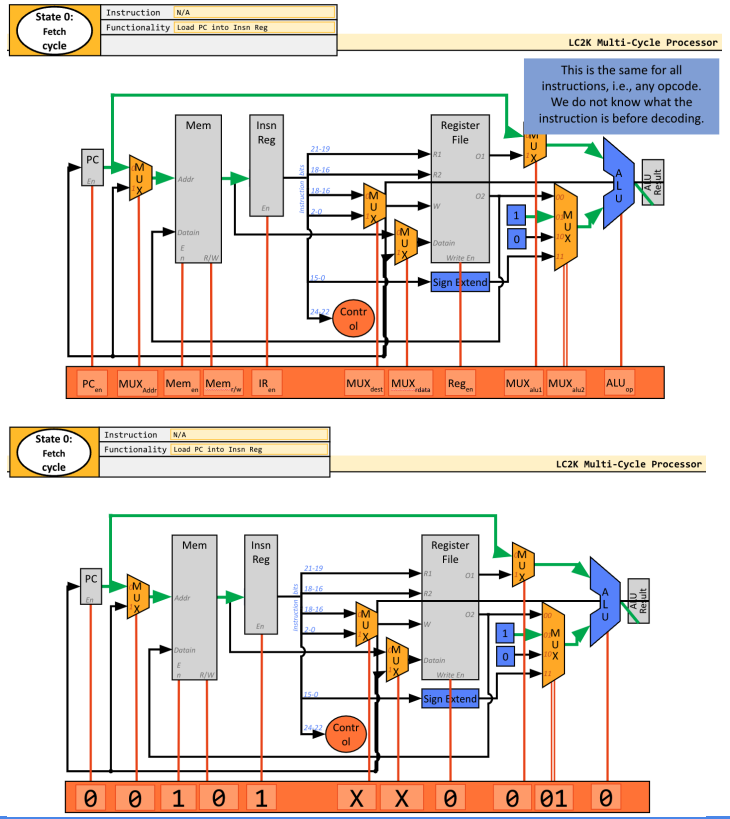


Multi Cycle Datapath

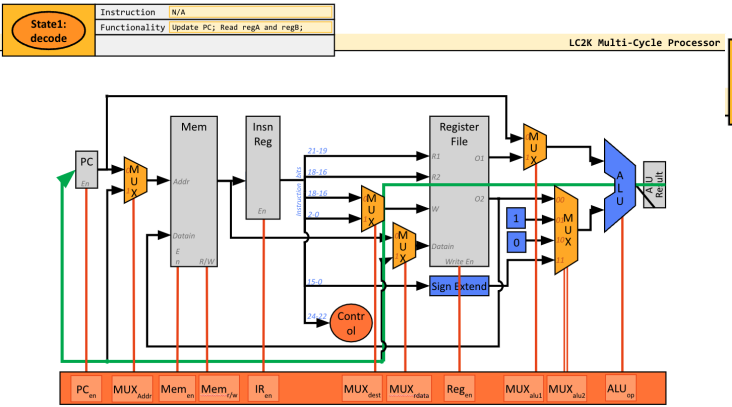
- Uses less components than single cycle datapath
- Different instructions take different time to run
- Cannot use two components sequentially in one cycle
  - e.g. “Calculate regA+regB in ALU **and then** write the result to destReg” == BAD

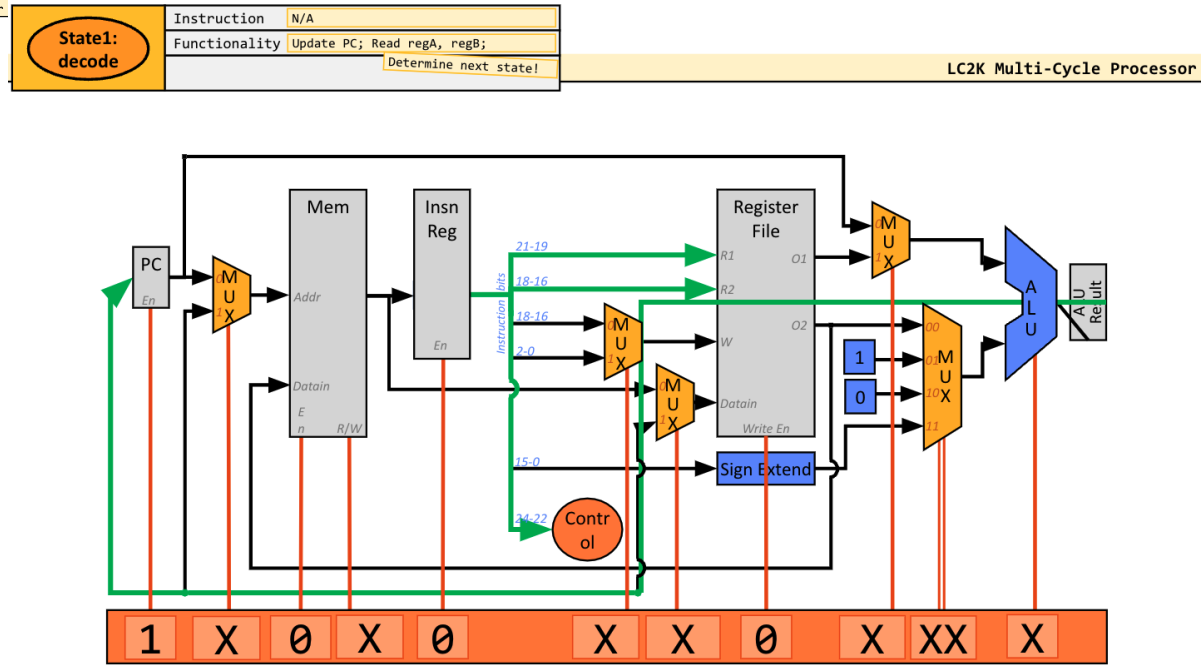
First Cycle (State 0) Fetch Instruction

- Read memory[PC] and store into instruction register.
  - Must select PC in memory address MUX (MUXaddr= 0) - *B in diagram*
  - Enable memory operation (Memen= 1) - *C in diagram*
  - R/W should be (read) (Memr/w= 0) – *D in diagram*
  - Enable Instruction Register write (IRen= 1) – *H in diagram*
- Calculate PC + 1
  - Send PC to ALU (MUXalu1 = 0) – *I in diagram*
  - Send 1 to ALU (MUXalu2 = 01) – *J in diagram*
  - Select ALU add operation (ALUop = 0) – *K in diagram*
- PCen = 0; Regen = 0; MUXdest and MUXrdata= X
  - *A, E, F, G in diagram, respectively*



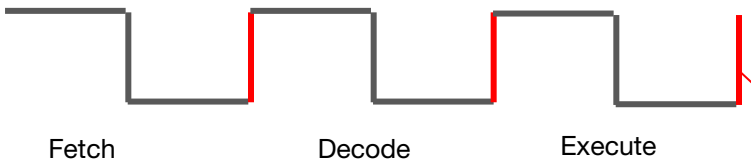
State 1: Decode



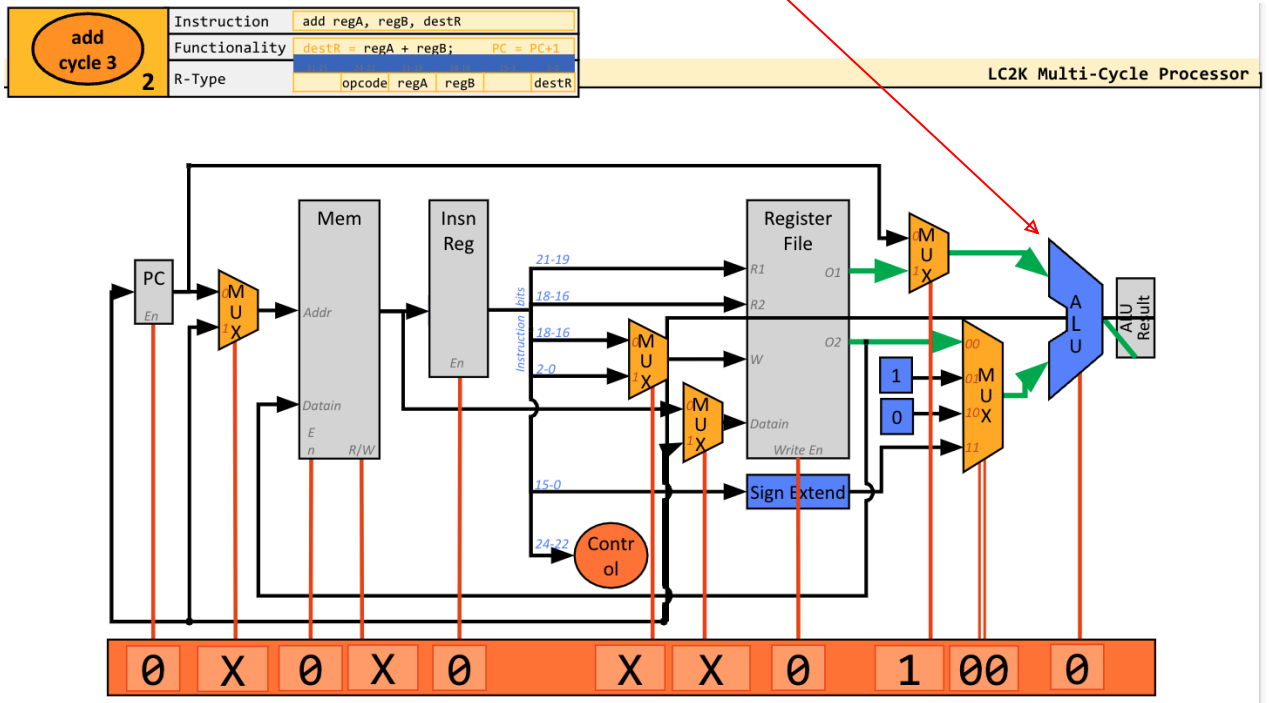


add 1 2 3

reg3 = reg1 + reg2



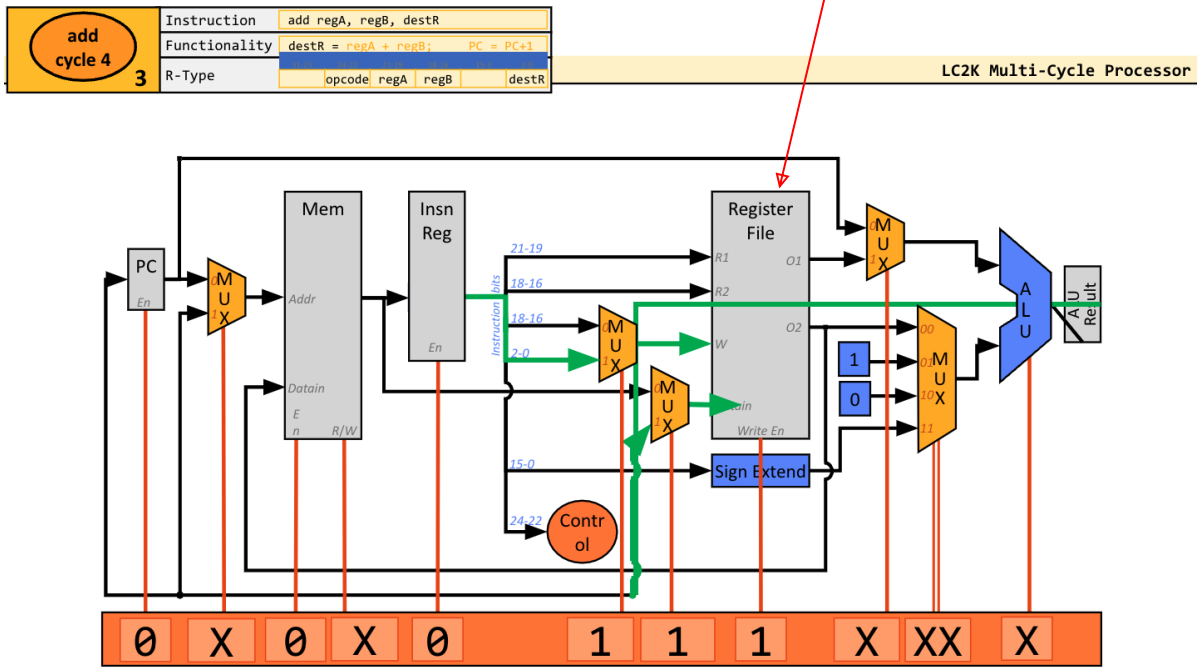
1. Fetch the instruction
2. Decode, read regs, update PC
3. ALU operation



add 1 2 3

reg3 = reg1 + reg2

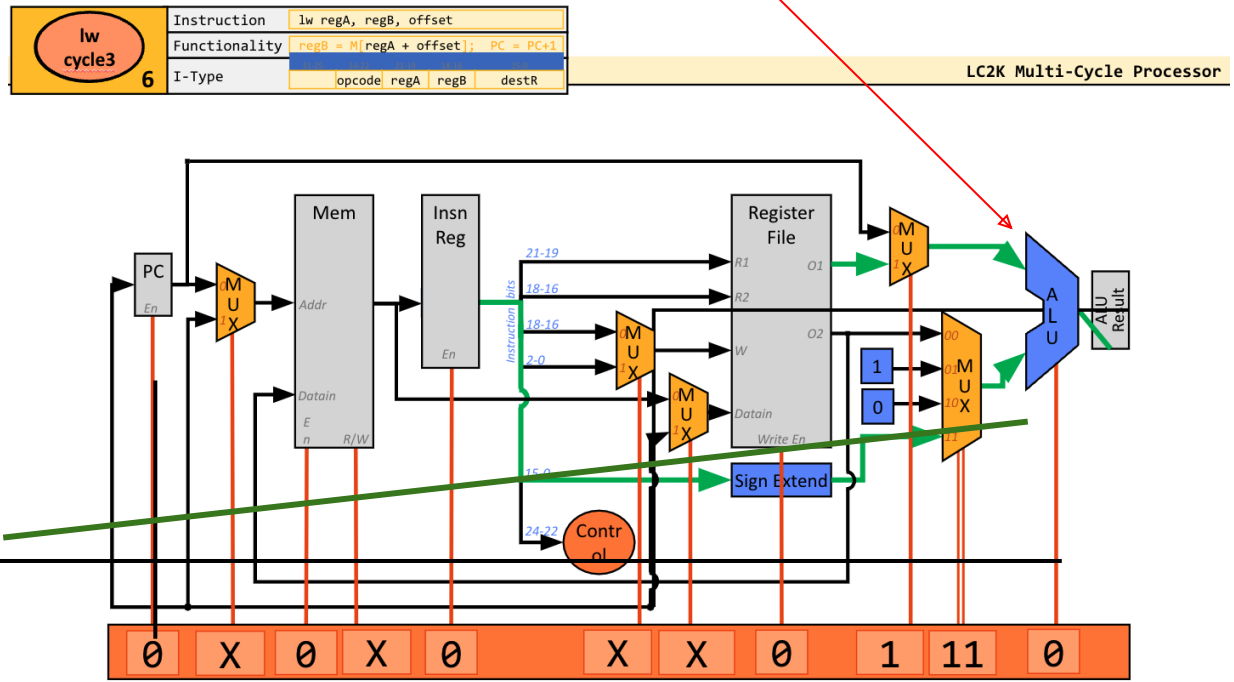
1. Fetch the instruction
2. Decode, read regs, update PC
3. ALU operation
4. Writeback to register file



lw 0 2 3

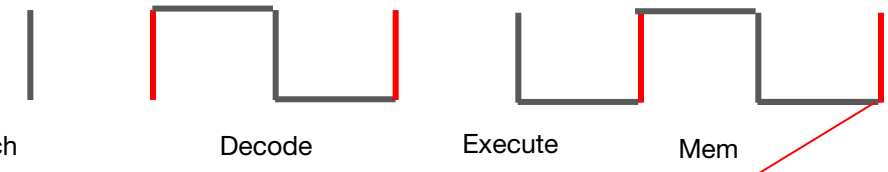
reg2 = Mem[reg0 + 3]

1. Fetch the instruction
2. Decode, read regs, update PC
3. ALU operation (regA + offset)

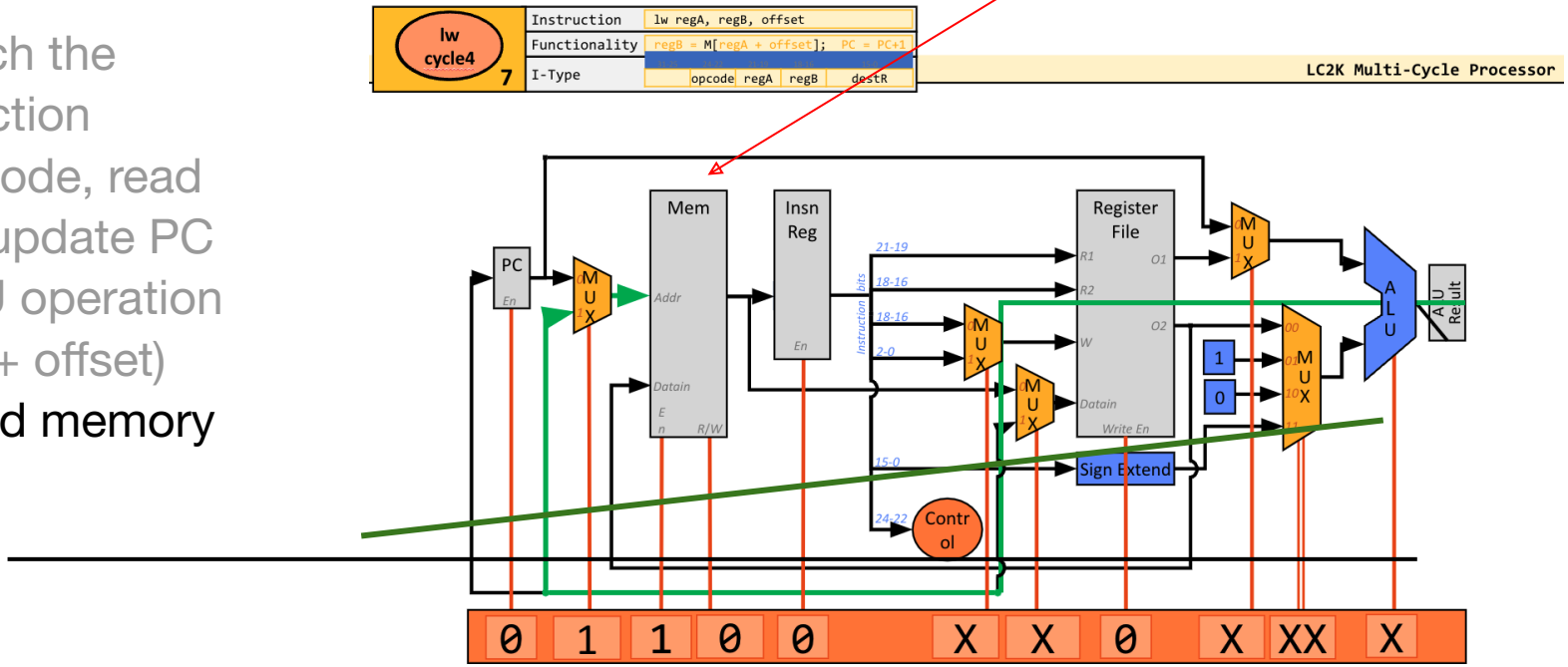


lw 0 2 3

reg2 = Mem[reg0 + 3]



1. Fetch the instruction
2. Decode, read regs, update PC
3. ALU operation (regA + offset)
4. Read memory

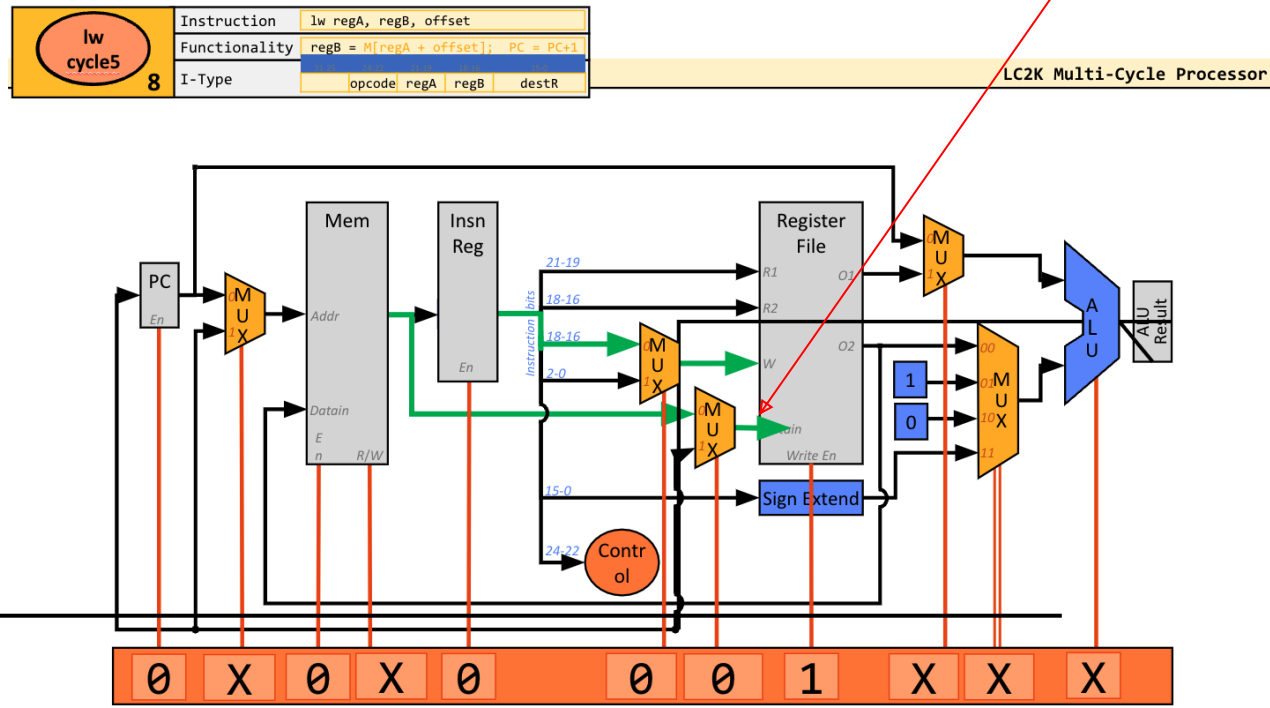


lw 0 2 3

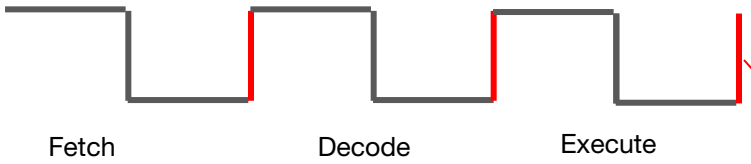
reg2 = Mem[reg0 + 3]

Fetch Decode Execute Mem Writeback

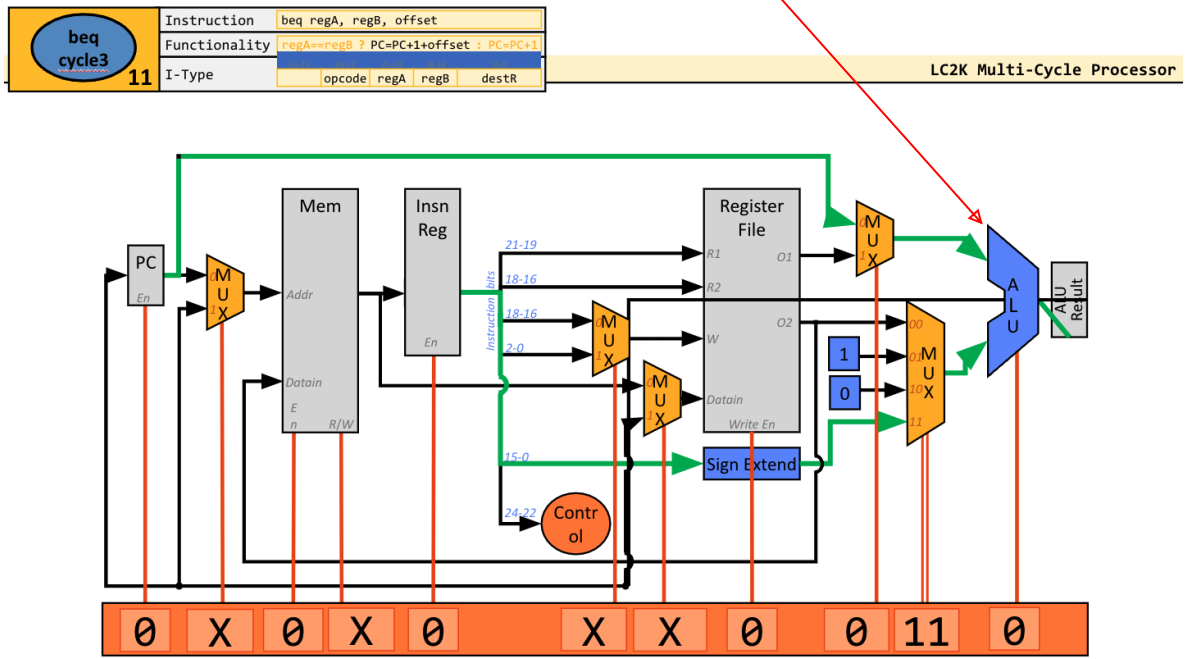
1. Fetch the instruction
2. Decode, read regs, update PC
3. ALU operation (regA + offset)
4. Read memory
5. Writeback



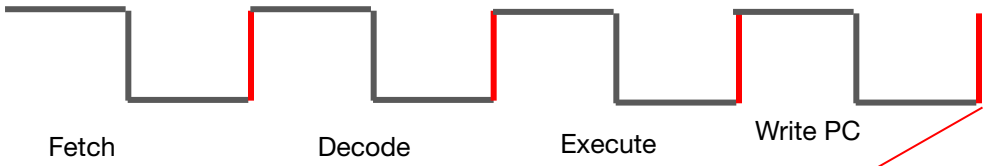
beq 0 0 0



1. Fetch the instruction
2. Decode, read regs, update PC
3. ALU operation (PC+offset+1)

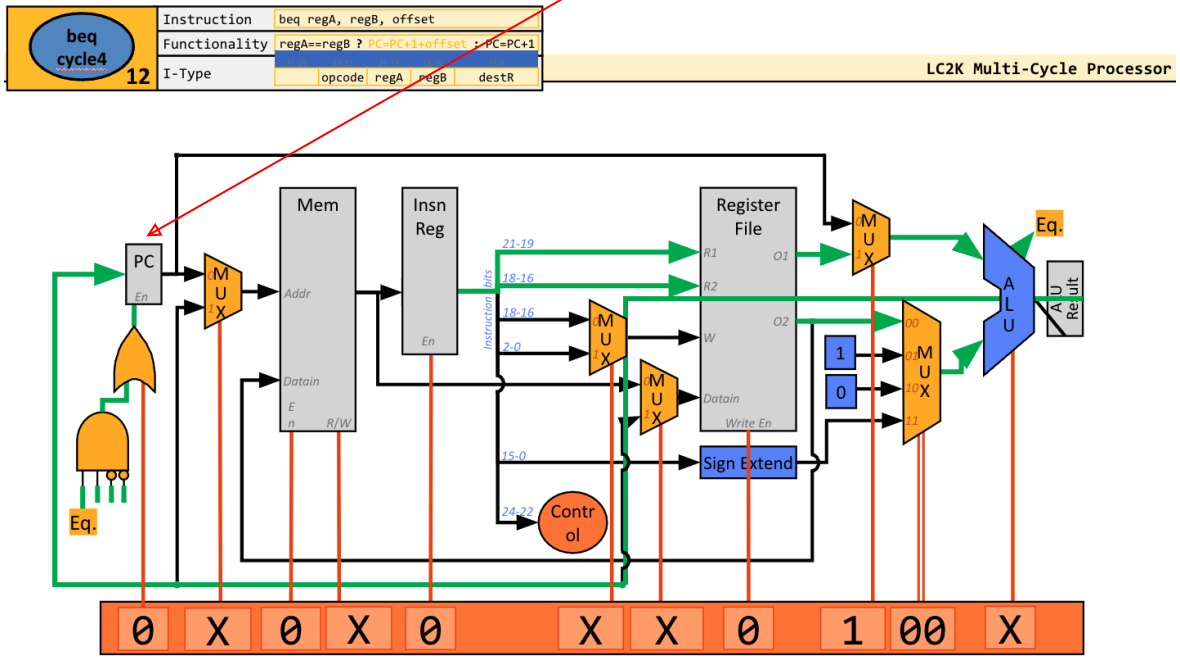


beq 0 0 0

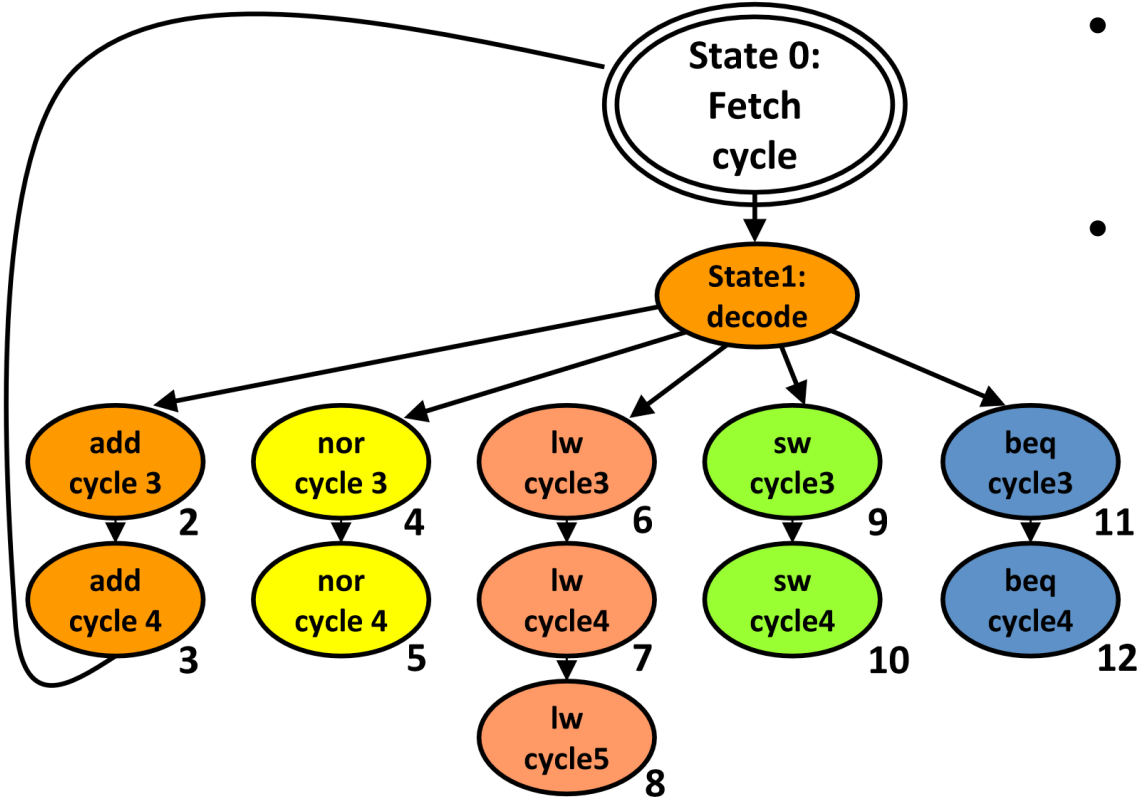


1. Fetch the instruction
2. Decode, read regs, update PC
3. ALU operation (PC+offset+1)
4. Write PC

Compare valA and valB

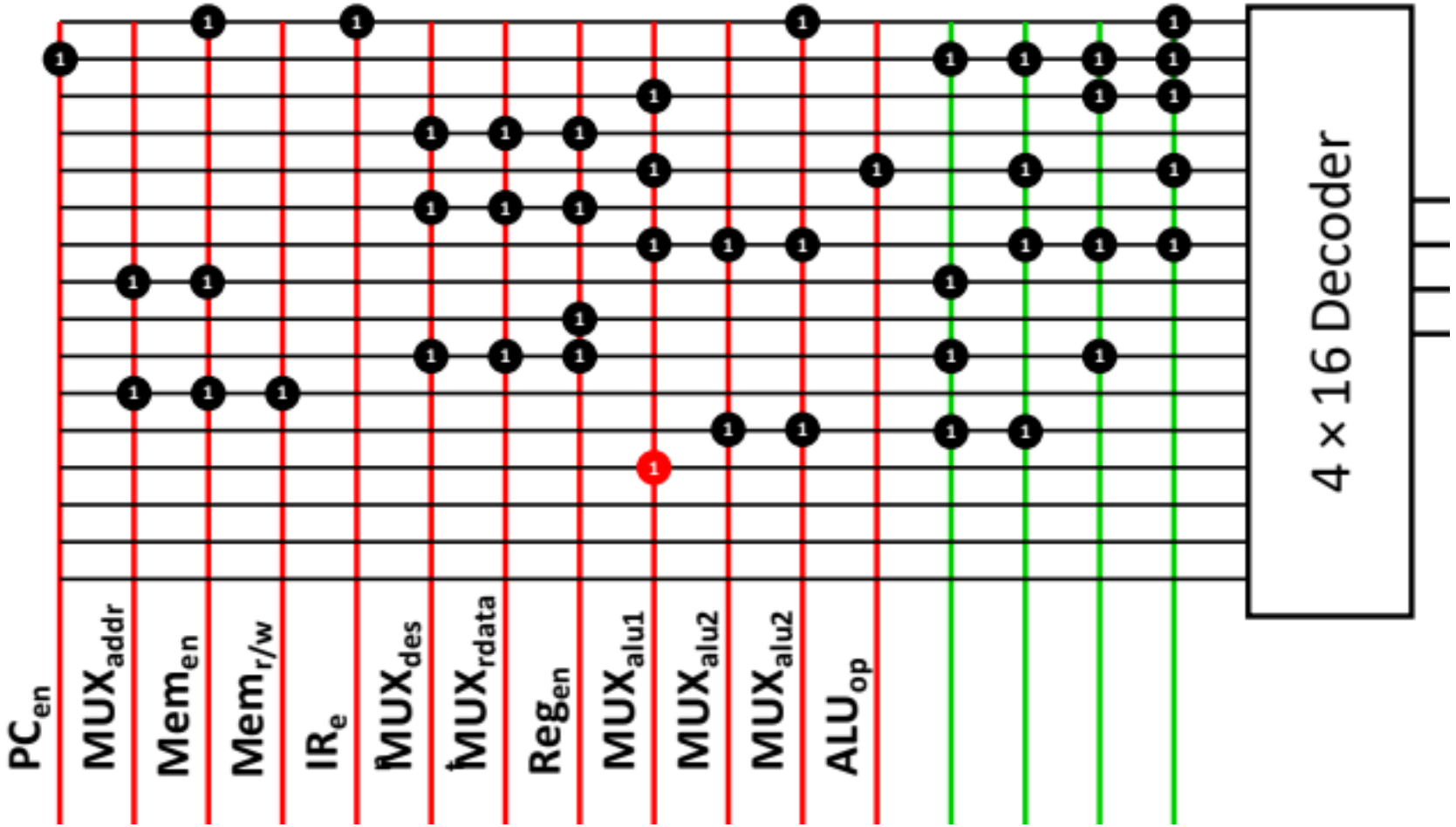


Multi Cycle Datapath



- Noop goes back to fetch (state 0) after decode (state 1)
- Halt stops in decode (state 1)

Still Us
Contro
ROM t
Determ
State



Multi Cycle Datapath

- Cycle period determined by slowest **component**
- Average CPI ~ 4 for most LC2K programs
  - We like to determine average CPIs for multicycle benchmarks

Performance analysis

$$\text{Performance} = \frac{\text{\# instruction}}{\text{\# program}} \text{X} \frac{\text{\# cycle}}{\text{\# instruction}} \text{X} \frac{\text{time}}{\text{\#cycle}}$$

Program size X CPI X Clock Period

Trade off between CPI and clock period

Problem 3: Multi-Cycle Datapath Calculations

We are using a benchmarking program that executes 2,000 instructions. 10% are `lw`, 25% `sw`, 45% `add` / `nor`, 10% `beq`, and 10% `noop` / `halt`.

Each use of the ALU costs **10ns**, accessing memory costs **30ns**, register file reads and writes cost **5ns**, and everything else costs **0ns**.

How many cycles are needed to complete this benchmark?

What is the total execution time of this benchmark?

What is the average CPI (Cycles per Instruction)?

Problem 3a: Multi-Cycle Cycles Calculation

We are using a benchmarking program that executes 2,000 instructions. 10% are **`lw`**, 25% **`sw`**, 45% **`add`** / **`nor`**, 10% **`beq`**, and 10% **`noop`** / **`halt`**.

How many cycles are needed to complete this benchmark?

2,000 × (0.25 (**`sw`**) + 0.45 (**`add`** / **`nor`**) + 0.10 (**`beq`**)) × 4 cycles / inst. +
2,000 × 0.1 (**`lw`**) × 5 cycles / inst. +
2,000 × 0.1 (**`noop`** / **`halt`**) × 2 cycles / inst.

= **7,800 cycles**

Problem 3b: Multi-Cycle Execution Calculation

Each use of the ALU costs **10ns**, accessing memory costs **30ns**, register file reads and writes cost **5ns**, and everything else costs **0ns**.

What is the total execution time of this benchmark?

Clock Cycle = Slowest *Component* (Memory) = 30ns

7,800 cycles ✕ 30ns / cycle =

234,000ns

Problem 3c: Multi-Cycle CPI Calculation

We are using a benchmarking program that executes 2,000 instructions. 10% are **`lw`**, 25% **`sw`**, 45% **`add`** / **`nor`**, 10% **`beq`**, and 10% **`noop`** / **`halt`**.

Each use of the ALU costs **5ns**, accessing memory costs **15ns**, register file reads and writes cost **2ns**, and everything else costs **0ns**.

What is the average CPI (Cycles per Instruction)?

7,800 cycles / 2,000 instructions = **3.9 cycles / instruction**

Datapath Comparison

The code is run on two different datapaths (A, B) to compare the performance of each. Please find out which datapath is single cycle and which is multi-cycle. And what is clock period for each datapath.

```
1          lw      0      1    data
2         lw      3      5    9
3       start    add   3      4    3
4           sw      2      2    0
5        beq    0    0    next
6     next    noop
7        halt
8     data    .fill    1234
```

Num. Completed instructions	A Time (ns)	B Time (ns)
1	200	125
2	400	250
3	560	375
4	720	500
5	880	625
6	960	750
7	1040	875

Datapath Comparison

The code is run on three different datapath (A, B, C) to compare the performance of each. Please find out which datapath is single cycle and which is multi-cycle. And what is clock period for each datapath.

```
1          lw      0      1     data
2         lw      3      5     9
3       start    add   3      4     3
4          sw      2      2     0
5        beq    0    0    next
6     next    noop
7        halt
8     data    .fill    1234
```

Num. Completed instructions	A Time (ns) Multi-Cycle Clock Period=40ns	B Time (ns) Single-Cycle Clock Period=125ns
1	200	125
2	400	250
3	560	375
4	720	500
5	880	625
6	960	750
7	1040	875

Topics Covered

- Understanding the LC2K ISA
- ARM
- Endianness
- Linking/Linker
- Caller/Callee
- LC2K Processor Performance
  - Single-Cycle Datapath
  - Multi-Cycle Datapath
- Pipeline Datapath
- **CheatSheet Stuff**

Cheat Sheet Stuff

- Definitions for T/F
  - Endianness
  - Two's Complement vs IEEE
    - Can all numbers be represented?
  - Stack vs Heap; where do variables go?
    - Stack
    - Heap
    - Static
    - Text
  - Function Calls - Main vs Leaf vs Intermediate - caller vs callee

Cheat Sheet Stuff

- Complete simple FSMs
  - Draw an FSM that accepts numbers with an odd number of 1s
- Formulas
  - Calculate size of ROM, Decoder, Controller
  - What's the size of the ROM given FSM states, input bits, and output bits
- Logic Gates
  - How to create them from NAND and NOR
  - How to create desired functionality using specified gates
  - Circuit Delay
- Struct Alignment
  - Always align based on largest primitive
  - Remember difference between 64 and 32 bit architecture
  - Know how to optimize alignment given variables in a struct

Cheat Sheet Stuff

- Caller Callee Tips
  - Do you store parameters? Do you store when main is called vs not?
- Linker/ Linking tips
  - Extern vs #define vs #include vs function calls
  - When do things need to be relinked?
  - When does linking go wrong?
- Simulating LC2k
  - Write down what each instruction does and how it affects registers
  - Difference between pointing to a line address and pointing to a value
  - Function Calls
  - Optimization tips for making LC2k code better
  - How does the stack work in LC2k?
  - Exposing bugs in LC2k code

Cheat Sheet Stuff

- LC2k Processor Performance
  - Single cycle vs multi cycle vs pipeline clock periods
  - Reverse engineering # of instructions based on # of cycles
  - Execution time given clock period
- Datapaths
  - Multicycle control ROM bits - how affected by new instructions
  - MUXes, know how to add them and their control bits
  - Complete cycles for multicycle datapath give cycle 1 and cycle 2 - how do these restrict your solutions?
- Test Case Knowledge
  - What makes a good test variable versus a bad one? I.e. What exposes a bug versus not

Cheat Sheet Stuff

- ISA Design
  - Address bits, byte vs word addressable
  - C to new ISA code - give yourself tips on translating C code
- C to ARM
  - Understand how to read the green sheet (given on the exam) and when to use certain opcodes

Cheat Sheet Stuff

IRON RULE: Execution Time = #Instructions*CPI*ClockPeriod = #Cycles*ClockPeriod

Clock Period

Single Cycle: Latency of Slowest Instruction
Multi Cycle: Latency of Slowest Cycle (generally slowest datapath component)

#Cycles

- Single Cycle: #Instructions
- Multi Cycle: #Instructions * sum over all opcodes(%opcodes*cycles for opcode)

CPI

Single Cycle: 1 (it's in the name)
Multi Cycle: #Cycles/#Instructions

#Cycles Per Opcode (MULTICYCLE ONLY)

- add/nor/sw/beq: 4 cycles
- lw: 5 cycles
- noop/halt: 2 cycles
- jalr: Don't worry about it